AN INTEGRATED COMPILER AND RUNTIME FRAMEWORK FOR SPARSE MATRIX CODES

by

Anand Venkat

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2016

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Anand Venkat**

has been approved by the following supervisory committee members:

Mary Hall ,	Chair(s)	**9 June 2016** Date Approved
Michelle Strout ,	Member	**9 June 2016** Date Approved
Matthew Might ,	Member	**9 June 2016** Date Approved
Ganesh Gopalakrishnan ,	Member	**9 June 2016** Date Approved
John Regehr ,	Member	**30 Nov 2016** Date Approved

by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Sparse matrix codes are found in numerous applications ranging from iterative numerical solvers to graph analytics. Achieving high performance on these codes has however been a significant challenge, mainly due to array access indirection, for example, of the form A[B[i]]. Indirect accesses make precise dependence analysis impossible at compile-time, and hence prevent many parallelizing and locality optimizing transformations from being applied. The expert user relies on manually written libraries to tailor the sparse code and data representations best suited to the target architecture from a general sparse matrix representation. However libraries have limited composability, address very specific optimization strategies, and have to be rewritten as new architectures emerge.

In this dissertation, we explore the use of the inspector/executor methodology to accomplish the code and data transformations to tailor high performance sparse matrix representations. We devise and embed abstractions for such inspector/executor transformations within a compiler framework so that they can be composed with a rich set of existing polyhedral compiler transformations to derive complex transformation sequences for high performance. We demonstrate the automatic generation of inspector/executor code, which orchestrates code and data transformations to derive high performance representations for the Sparse Matrix Vector Multiply kernel in particular. We also show how the same transformations may be integrated into sparse matrix and graph applications such as Sparse Matrix Matrix Multiply and Stochastic Gradient Descent, respectively. The specific constraints of these applications, such as problem size and dependence structure, necessitate unique sparse matrix representations that can be realized using our transformations.

Computations such as Gauss Seidel, with loop carried dependences at the outer most loop necessitate different strategies for high performance. Specifically, we organize the computation into level sets or wavefronts of irregular size, such that iterations of a wavefront may be scheduled in parallel but different wavefronts have to be synchronized. We

demonstrate automatic code generation of high performance inspectors that do explicit dependence testing and level set construction at runtime, as well as high performance executors, which are the actual parallelized computations.

For the above sparse matrix applications, we automatically generate inspector/executor code comparable in performance to manually tuned libraries.

For my parents

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Sparse matrix applications are found in a wide range of applications including iterative methods for solving large sparse linear systems, molecular dynamics, and large-scale graph applications. Sparse matrix representations only store the nonzero values of the matrix, and have additional auxiliary arrays to record the row and/or column positions. Attaining high performance on sparse matrix applications is difficult due to the extra memory accesses for the auxiliary arrays. These computations are memory bound and only achieve a small fraction of the peak memory bandwidth. Further, caches are not used effectively in these computations due to their irregular access patterns. Due to these factors, one of the frequently used kernels in sparse applications, Sparse Matrix Vector (SpMV) multiply has typically run at 10% or less of the peak performance of the machine [2].

Libraries are commonly used to specialize for the nonzero structure of the matrix and the underlying architecture to optimize the performance for these applications. However library based solutions tend to be fixed-function or address a very specific application. Further, the code has to be manually optimized for new representations on emergent architectures [3].

Compiler-based approaches, on the other hand, though having the strength of composability of optimizations have found sparse applications difficult to optimize as runtime information is required to understand the memory access patterns and dependences of these applications.

In this dissertation we utilize runtime information to either (i) unravel the memory locations being accessed; (ii) modify the computation by reorganizing the code's iteration space; and (iii) accomplish an additional data transformation on associated data to optimize for parallelism and/or data locality. Specifically we target the automatic generation

of such high performance runtime code as well as the composability of such transformations with other existing compiler transformations in a polyhedral compiler framework to optimize sparse matrix codes.

1.1 Existing strategies for high performance of sparse matrix codes

No single sparse matrix representation is optimal given a range of inputs and architectures; identifying the best performing solution requires careful input and architecture specific tuning. To address this issue, many hand-written libraries provide a range of sparse matrix representations that the user may choose to convert to, starting from a standard format such as Compressed Sparse Row (CSR) or Coordinate (COO) as illustrated in Figure 1.1. In addition to deriving a modified sparse representation libraries are also utilized to reorganize the computation at runtime for parallelism once the dependences are known. These optimizations may also be realized using inspector/executor methods, where an inspector does the conversion of the matrix to the desired representation and/or reorganizes the computation for parallelism, and the executor is the optimized computation modified to refer to the new representation. Compiler approaches for sparse matrix computations use modified representations to mimic the dense computation, so that dependences can be extracted easily, and rely on additional transformations to convert from a dense to a sparse representation.

1.1.1 Libraries

One of the common strategies used by libraries in deriving an optimized sparse matrix format is reducing the indexing overhead associated with the input matrix. This can improve the memory performance of the resulting application.

The optimizations applied by the Optimized Sparse Kernel Interface (OSKI) [4] library includes specializing the SpMV application for matrices with block substructure, enforcing the structure at times by inserting nonzeros, so that the same set of adjacent rows and columns may be reused or cached in registers for high performance. Though this strategy is fixed across architectures, the best 2D block size for performance for a given architecture is nonobvious. OSKI generates multiple code variants for different block sizes and then applies autotuning techniques to identify the best configuration on the target architecture.

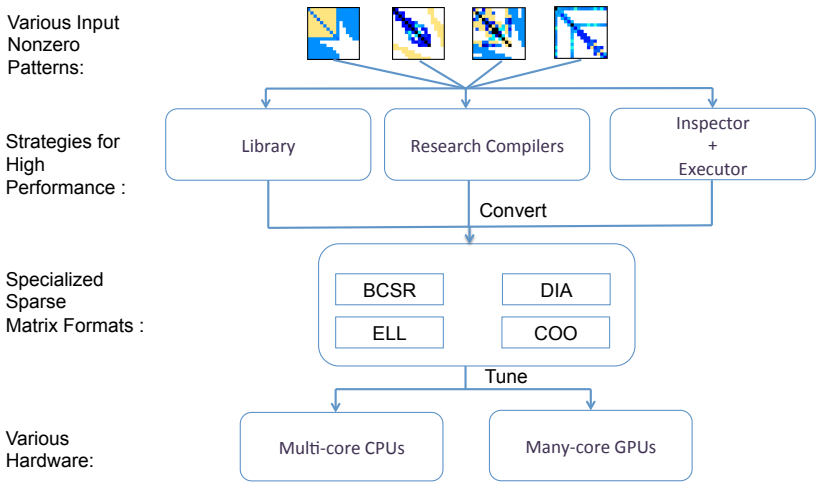


Figure 1.1: Existing approaches for high performance sparse applications. Sparse aatrix figures adapted from *A library of auto tuned sparse matrix kernels for many core processors* by W. Abu-Sufah, 2015 [1].

NVIDIA's library for sparse linear algebra and graph computations, CUSP [5], specializes for SpMV with formats that exploit coalesced memory accesses to their Graphics Processing Unit (GPU) global memory subsystem or take advantage of low latency shared memory to maximize throughput. OSKI and CUSP are examples of the manually tuned libraries used by programmers to achieve high performance on sparse applications.

Achieving high performance on sparse matrix applications requires careful architecture-specific tuning and as new architectures emerge, these libraries would have to be upgraded to keep abreast of the architectural advancements.

1.1.2 Inspector/executor

Inspector/executor transformations are a class of runtime transformations, where the *inspector* is the code that may traverse the original code's iteration space, or analyze certain index arrays and potentially reorder or restructure the code. Additionally it may reorganize the data referenced by the code. The *executor* is the optimized version of the original computation that references the potentially reordered iteration space and/or memory references by the inspector code.

Inspector/executor transformations have been utilized in isolation in the past to carry out parallelization, iteration space reordering, and data transformation [6–14]. These iteration and data reordering transformations can facilitate the derivation of specialized sparse

matrix representations similar to a library-based approach.

The work due to Strout et al. [15] utilizes a compiler to compose various runtime transformations such as consecutive packing, lexicographic grouping, graph partitioning, and full sparse tiling [16]. We espouse the same philosophy of composing these transformations within a compiler framework and extend polyhedral code generation to accommodate these transformations in an end-to-end compiler transformation and code generation framework.

1.1.3 Compiler approaches

To circumvent the problem of indirect accesses, some compiler approaches [17–19] start with a dense abstraction of the computation and rely on the underlying compiler, along with user-supplied directives, to generate the analogous sparse representation of the code during code generation. Typically these compilers are limited in that they either incorporate a small, fixed set of matrix representations for which code generation is straightforward and/or rely on the user to provide implementations for accessing data in sparse formats for operations such as searching, enumeration, and dereferencing [3]. These facilitate the conversion from dense to sparse code, making the compiler responsible for optimizing the overhead of the dense code. Furthermore, some of these solutions require the user to specify the input nonzero pattern of the sparse matrix in order to derive the specialized implementation. In almost all these cases, a comparison of running times of the dense to sparse conversion with manually tuned libraries was not demonstrated. A key feature of our work is the automatic generation of high performance inspector codes that perform this dense to sparse conversion.

1.2 Targets of compiler optimization

We implement our transformations for sparse matrix codes in a polyhedral compiler for their ease of composing transformations. Polyhedral compilers have proven to be powerful in optimizing regular affine codes. There have been a number of polyhedral transformation and code generation frameworks that have been successful in optimizing dense array codes [20–23]. However they have been severely limited in applicability to codes with nonaffine code constructs such as array indirection.

Since inspector/executor transformations are an effective runtime methodology to reorganize a computation for better locality and/or parallelism, providing support for such transformations within a polyhedral compiler framework and allowing nonaffine transformations to be composed with regular affine transformations could reveal interesting opportunities for a compiler based optimization framework for sparse codes. Such an approach would have implications for all aspects of polyhedral compilation: iteration space representation, code transformation, dependence analysis, and code generation. This dissertation touches on all these aspects and extends a real system for a holistic approach to embedding inspector/executor transformations in a polyhedral context.

The twin objectives of our work are automatically generating high performance inspector/executor code transformations from a compile-time specification and composing these transformations with other common compiler transformations. This necessitates abstractions to represent nonaffine code constructs as well as embed runtime inspector/executor functionality as nonaffine code transformations. Our compiler framework was extended for this purpose and these abstractions were used for iteration and data representation and reorganization and dependence information extraction, all involving nonaffine code constructs.

The inspector/executor transformations target codes with nonaffine code constructs and reorganize both the code and associated data for potentially better parallelism, memory reuse, and load balance. Where dependences obstruct full parallelization, the iteration space is organized into wavefronts whose relative order of execution will respect dependences but have better performance than the original sequential version of the code.

1.2.1 Code transformation

Codes with nonaffine loop bounds and/or array subscripts are transformed within our framework in a number of ways for high performance. Accommodating and manipulating such nonaffine constructs can pave the way for a number of code optimizations. For example, representing nonaffine loop bounds enables many other loop transformations such as tiling to be applied on the specified loop. We also present a generalized loop coalescing transformation that can tolerate codes with nonaffine loop bounds and convert a loop of multiple loop levels to a single loop level, recording the correspondence between

the output loop and input loop levels via runtime inspection in the process. The runtime functionality of the transformation is itself abstracted as a nonaffine transformation. Our approach demonstrates how inspector/executor code transformations may be integrated as nonaffine transformations in a compiler framework.

Further, code transformations that convert between the *dense* and sparse versions of the computation are utilized: a *dense* conversion uncovers the analogous dense iteration space of a sparse computation, thus introducing affine loop bounds at the expense of redundant iterations. Other affine transformations manipulate these affine loop bounds, and hence reorder the iteration space to facilitate the derivation of a desired representation. Once all intermediate transformations have manipulated the affine bounds introduced, a sparse iteration space is derived from the dense one with the redundancy introduced earlier being eliminated. These code transformations each have an automatically generated inspector/executor code component.

Incorporating abstractions for nonaffine code constructs and encapsulating inspector/executor code transformations as nonaffine transformations in a compiler framework allowed us to compose these transformations with other existing polyhedral transformations with ease and derive complex transformation sequences for high performance code.

1.2.2 Data transformation

In some cases, introducing a small number of zero valued elements to the computation can result in more regular memory accesses and better performance. For example, for the Block CSR (BCSR) sparse matrix representation utilized in OSKI [4], this strategy is used so that adjacent rows and columns belonging to the same block may be stored with less indexing overhead per nonzero and also because they may be cached and reused from low latency memory.

For this purpose data transformations are introduced, taking a user specified array and loop levels to consider for footprint as input. Inspector/executor code is then automatically generated to accomplish the data transformation according to specification. These data transformations are effected in conjunction with the code transformations that convert between the dense and sparse versions of the computation.

The above code and data transformations were utilized for the Sparse Matrix Vec-

tor(SpMV) Multiply kernel, where the combination of the newly introduced inspector/executor transformations with regular affine transformations was able to derive high performing code variants comparable in performance to manually tuned libraries such as CUSP on GPUs and OSKI on CPUs. The transformations enabled the derivation of SpMV based on Coordinate (COO), Diagonal (DIA), ELL, and Block-CSR (BCSR) starting from one based off the standard CSR representation.

1.2.3 Wavefront parallelization

Parallelizing the outermost loop of a SpMV computation is relatively straightforward due to the absence of cross-iteration dependences. Some important sparse matrix computations such as symmetric Gauss Seidel relaxation and Incomplete LU0 (ILU0) factorization, which are used as preconditioners for the Preconditioned Conjugate Gradient (PCG) algorithm, have loop-carried dependences at the outer most loop level, thus requiring paralllelization with synchronization. ILU0 factorization is computed once to find L and Umatrices such that $L * U$ is close to the input matrix A, when higher powers of A are used as preconditioners such as A^1,A^2, the computation is termed as ILU(1) and ILU(2), respectively.

For these types of computations, we derive the cross-iteration dependence constraints at compile-time, and supply them to a runtime inspector that explicitly constructs the dependence graph.

The dependence graph is then traversed in a topological order. This traversal produces a partial order of wavefronts; a wavefront consists of iterations that can execute in parallel, and the partial order captures dependences among wavefronts. The approach of parallelizing within wavefronts, with synchronization between wavefronts, is wavefront parallelism.

We integrate the inspector that constructs the dependence graph with a library implementation for deriving wavefronts, and these are integrated with a compiler-generated executor, which is the parallelized computation. We demonstrate the effectiveness of this parallelization of both the inspector and executor code, over sequential execution for a preconditioned conjugate gradient algorithm (PCG).

1.2.4 Application integration

We demonstrate how the code and data transformations we introduce can be adapted and integrated into applications requiring highly specialized sparse representations. Specifically, our compiler transformations can derive the Sparse Matrix-Matrix Multiply computation using the Compressed Sparse Block (CSB) [24] used recently in a block eigensolver, Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) for very large symmetric sparse matrices having 100s of million nonzeros. The CSB format conceptually blocks the columns and rows of the input matrix into square tiles and stores the nonzeros falling in each tile in the Coordinate(COO) format. We were able to derive this format using our code and data transformations for converting between dense and sparse representations and composing them with tiling. The compiler generated code matched and, in some cases, exceeded the performance of the manually written SpMM [24] code.

In recent work, we also demonstrated how code and data transformation techniques could be applied to a graph algorithm Stochastic Gradient Descent, where the graph is stored in a CSR format. The computation has a unique dependence pattern where nonzeros that share the same row or column cannot be processed concurrently. Hence, the same code and data transformations were used to derive diagonal and blocked diagonal conflict-free, schedules of the computation [25].

1.3 Thesis contributions and organization

Figure 1.2 presents an overview of our compiler and runtime framework. Codes potentially containing nonaffine loop bounds or array subscripts are analyzed, and an initial dependence extraction that tolerates these code constructs is done. The dependences constrain the sequence of code restructuring affine and inspector/executor transformations abstracted as nonaffine transformations within the CHiLL/CUDA-CHiLL polyhedral transformation framework. These nonaffine transformations can be composed with existing affine transformations in CHiLL. CUDA-CHiLL is a thin wrapper around CHiLL and supports CUDA code transformations.

CHiLL internally relies on the affine and nonaffine abstractions exported by Omega+ as well as the simplification of nonaffine relations offered by IEGenLib. Once the iteration spaces and transformations have been prepared in set and relation form respectively by

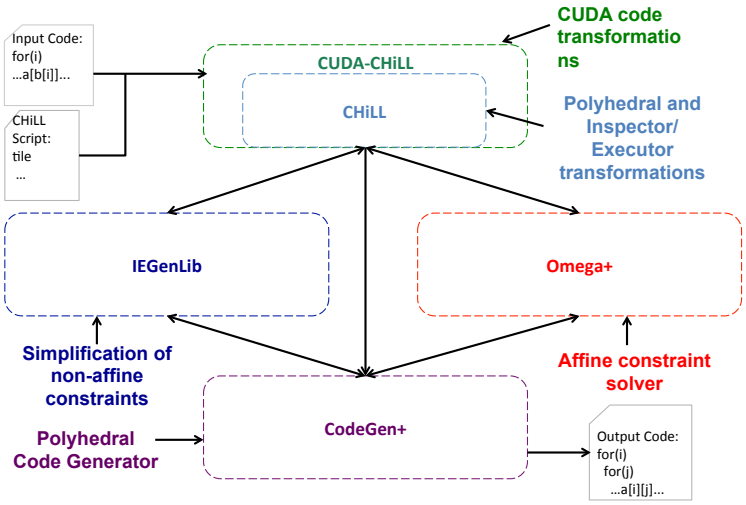


Figure 1.2: Overview of compiler and framework.

CHiLL, they are passed to the polyhedra scanning code generator CodeGen+. CodeGen+ has been extended to handle and manipulate nonaffine abstractions and is capable of generating inspector/executor code according to the specifications from CHiLL.

A key advantage of our approach over existing library-based techniques is that the same set of core inspector-executor transformations were applicable in a variety of contexts yielding high performance code applications, illustrating the power of a compiler-based approach. This is significant in the context of optimizing sparse matrix applications where attaining high performance depends on a complex combination of the characteristics of the input matrix, the particular application as well as the specific architecture, hence necessitating the provision of multiple optimization capabilities by the compiler that addresses these three requirements, as well as their ease of composition.

1.3.1 Contributions

1. Developing the first end-to-end polyhedral transformation and code generation system that accommodates and transforms codes with nonaffine loop bounds and array subscripts, including inspector/executor transformations represented as nonaffine transformations. Specifically the *generalized* loop coalescing transformation is introduced as a nonaffine transformation that converts a loop of multiple dimensions to

a loop of a single dimension.

2. Development of new compiler transformations, *make-dense*, *compact*, and *compact-and-pad*, for codes with indirection through index arrays that facilitate code and data transformations. Code transformations include converting a loop of multiple dimensions and data transformations include padding and insertion of zero elements. The code and data transformations direct the automatic generation of high performance inspector/executor codes whose performance is competitive with manually tuned libraries. These new transformations also compose with existing polyhedral transformations.

3. Integration of these new transformations to derive highly specialized representations for sparse matrix applications such as LOBPCG and SGD.

4. Automated dependence testing, simplification, and inspector/executor code generation for codes with loop carried dependences resulting in parallel, high performance inspector, and wavefront-parallel executor codes.

1.3.2 Thesis organization

A brief background on polyhedral compiler technology, inspector/executor methodology, various common SpMV representations, and the sparse matrix applications that were optimized in this thesis are presented in Chapter 2. The abstractions for nonaffine code constructs and transformations and how they enable high performance GPU code variant generation for SpMV are explained in Chapters 3 and 4. Chapter 5 describes new compiler code and data transformations that facilitate conversion from sparse to dense versions of the code and vice-versa, as well as their composition with existing polyhedral transformations to derive various high performance sparse representations. The automatic generation of the transformations corresponding inspectors is also discussed in this chapter.

Chapter 6 discusses the derivation of more advanced sparse matrix representations and their integration into two sparse matrix applications, LOBPCG and SGD. Chapter 7 presents the automated wavefront parallelization effected via a combination of compiler and runtime inspector/executor transformations for codes with loop carried or cross-iteration

dependences. Chapter 8 discusses related work, and Chapter 9 touches on future directions and concludes.

CHAPTER 2

BACKGROUND

This thesis presents extensions to polyhedral compiler technology for sparse matrix applications. Nonaffine code constructs, such as indirect accesses via an index array, eg.A[B[i]], are represented using uninterpreted functions, where uninterpreted denotes that the mapping from storage location to value is not analyzable.

We use inspector/executor run-time transformations to do code restructuring or iteration space reordering, additional data transformations and finally inspect dependences and reorganize computations for partial parallelism. These inspector executor transformations can derive a range of sparse matrix representations starting from a standard Compressed Sparse Row (CSR) representation. Uninterpreted functions are used to abstract the inspector/executor transformations as nonaffine functions or mappings of the iteration space. This chapter presents a brief background on polyhedral compilers, inspector/executor methods, common sparse matrix representations, and the applications that were optimized for this thesis.

2.1 Polyhedral transformations and code generation

This section describes the components that constitute a polyhedral compilation system. Integral aspects of this system include representing the iteration space of the code as a set of inequalities, applying multiple transformations in sequence on this iteration space, and utilizing polyhedral code generation techniques on the final transformed iteration space. We also briefly describe research on extending polyhedral compilers for nonaffine codes and transformations.

To illustrate the processes involved in a polyhedral compiler, let us consider the loop shifting transformation applied to the loop nest in Figure 2.1.

We utilize the Composable High Level Loop (CHiLL) polyhedral transformation framework to apply our source-to-source code optimizations. CHiLL harnesses both polyhedral

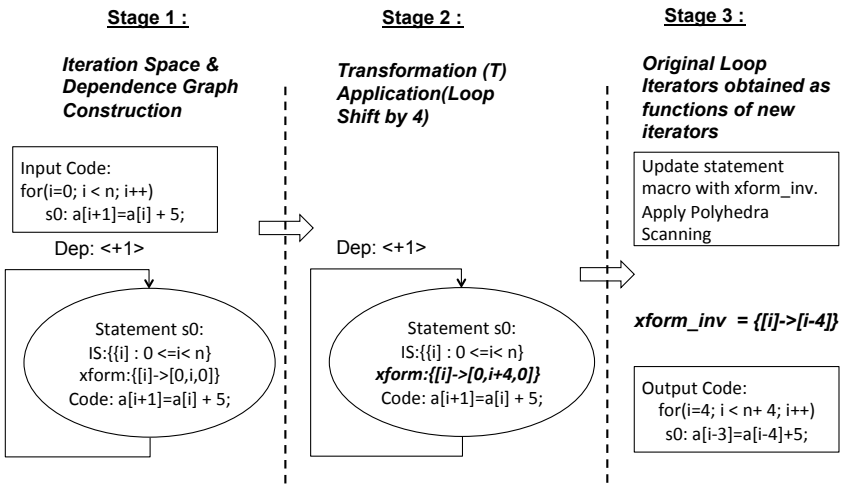


Figure 2.1: Polyhedral transformations and code generation.

and AST abstractions for its internal representation of a loop nest computation. The *Statement* is a central data structure to CHiLL's internal representation that achieves a clean separation between polyhedral and AST abstractions. Each statement in the code has three components:

- *IS*: The iteration space of a statement in a loop nest, represented as an integer tuple.

- *xform*: The transformation applied to this *IS*, represented as a mapping that takes the integer tuple as input and returns an output integer tuple.

- *code*: An actual pointer to the AST segment of the code. Hence, loop and conditional code constructs are created in polyhedral relation form, while the AST is encapsulated within the *code* field within the statement.

The code generator employs polyhedral scanning of the resulting iteration space to generate the output code shown. The original statement is updated with the inverse of the transformation mapping, that is, *xform_inv*, when updating the array access expressions. For instance, in Figure 2.1, we see the occurences of the original loop index *i* being replaced by the inverse of the transformation mapping *i-4* in the subscripts of array *a*.

CHiLL also represents dependences between statements in a dependence graph, illustrated as *Dep* in Figure 2.1. These data structures and their roles in transformation and code generation are described in this section.

2.1.1 Iteration space

Polyhedral frameworks describe the iteration space for each statement in a loop nest as a set of lattice points of a polyhedron [26]. For instance, we observe in Figure 2.2 the iteration space of the double-nested for loop represented pictorially as a 2-dimensional lattice as well as the inequalities that describe the iteration space.

Polyhedral frameworks such as Omega+, ISL, CLooG, PLuTo [21, 27–29] thus conveniently describe the iteration space as a set of inequalities over the set of loop variables utilizing this polyhedral abstraction. Additionally, all loop variable coefficients and constants are integers, since we need to only consider the integer space for the purposes of loop iteration spaces.

2.1.2 Transformations

Loop transformations can be viewed as mapping functions that convert the original iteration space to a transformed iteration space. For instance, the loop shifting transformation is encoded as a function that translates each point in the iteration space to the right by 4 units in *xform* in Figure 2.1.

This provides the compiler a powerful abstraction to transform a loop nest without being restricted to the original loop structure [30]. The bijective property of transformations also allows the output of one transformation to be composed as the input of the next transformation in complex sequences, deferring code generation until all transformations have been applied. Thus, a key strength of the polyhedral transformation approach is this

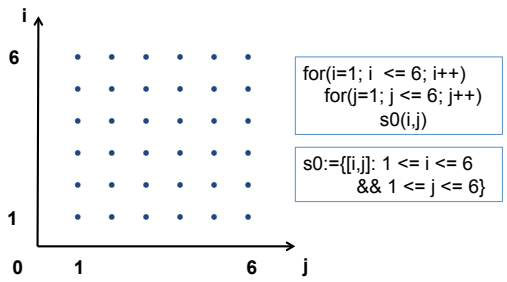


Figure 2.2: Iteration space as polyhedron.

ability to compose sequences of transformations.

The transformational mappings have also been termed as scattering functions in the literature [29]. They are one-to-one functions of the iteration space: the image of each point in the iteration space is unique under transformation. CHiLL [20] supports a range of loop transformations including loop distribution, fusion, tiling, unroll-and-jam, data-copy, index-set splitting, and permute. Also, CHiLL relies on Omega+ to construct the transformation mappings internally for each transformation, only exposing a high level scripting interface for the user to invoke the transformations.

2.1.3 Dependences

To verify the correctness of these iteration space remappings, the compiler employs *dependence analysis*, which detects possible accesses to the same memory location, where one of the accesses is a write. True dependences arise because of an initial write to a memory location and a subsequent read from the same location. Antidependences arise because of an initial read from a memory location and a subsequent write to the same location and output dependences arise due to an initial write to a memory location followed by a subsequent write to the same memory location.

A reordering of a statement's execution order is valid as long as it preserves all data dependences, that is, the source is always executed before the sink after transformation for any data dependence [31]. CHiLL constructs the iteration space and array access functions in relation form prior to utilizing Omega+ to apply the Omega test [32], to determine constraints for a dependence to hold for any pair of array references, at least one of which is a write.

Let us consider the example in Figure 2.1. The array subscript expression on the write and read are a[i+1] and a[i], respectively. Given the iteration space constraints the condition for this dependence is represented succinctly as follows:

$$I = \{[i+1] \rightarrow [i] \mid 0 <= i < N\} \tag{2.1}$$

To compute the dependence, the Omega test is employed by calling the *Deltas* function with this relation as input in the Omega library where *Deltas* is according to Definition 1.

Definition 1. *For a relation r, such that,* $r = \{x_1 \rightarrow y_1 | f(x_1, y_1)\}$ *the result of* ***Deltas(r)*** *is defined as* $\{z | \exists x, y \; s.t. \; f(x, y) \wedge z = y - x\}$

Informally, *Deltas* computes the dependence distance or the difference in iterations, between the source and sink of the dependence. For this example *Deltas* would compute a dependence distance of 1.

CHiLL constructs a dependence graph with each statement assuming a vertex and statements with dependences between them are connected with an edge, which contains information as to the type of dependence (true, anti, output) and dependence distance [33]. In Figure 2.1 we see that the dependence distance is +1 represented by the edge from statement s0 to itself.

As each transformation is applied, CHiLL checks for any dependence violation by traversing the dependence graph, and checking the semantics of the particular transformation; it throws an exception if a violation occurs. It also incrementally updates the dependence graph after each transformation. The legality tests are outlined in [20] and in general check for a loop carried dependence that is negative, that is, the sink of a dependence occurring before the source, which is a dependence violation.

2.1.4 Code generation

The bijective property of transformations is utilized in polyhedra scanning code generators [22, 34–37] to update array reference expressions (array access functions [38]) containing the loop indices with the unique inverse of the one-to-one transformational mapping as indicated by *xform_inv* in Figure 2.1.

Projection is a common operation that is used in polyhedra scanning code generators and is described in Definition 3.

Definition 2. *For a given set of constraints S on a set of variables* $I_1, \ldots, I_n$*, Project(S,* I_j*) is defined as the set of reduced constraints obtained from S by eliminating every occurrence of variable* I_j *from all (in)equalities in S using Fourier-Motzkin elimination, that is, every pair of inequalities of the form* $lb \leq c_1 I_j, c_2 I_j \leq ub$ *is replaced by a new inequality* $c_2 lb \leq c_1 ub$*.*

Given a loop nest, the loop index variables are projected from the innermost loop to the outermost loop, thus deriving the loop bounds on each variable. This is the first step that

most polyhedra scanning code generators employ. Each loop variable is a dimension in the polyhedron and, the bounds on each dimension are scanned in this fashion. CHiLL passes the final mappings and iteration space in relation and set form, respectively, to CodeGen+ which relies on an improved polyhedra scanning algorithm [34] to handle control-flow minimized code.

$$I = \{[i,j] : \mid 0 <= j \;\&\&\; j < i < N\} \tag{2.2}$$

For instance, let us consider the set of inequalities above, which are input to the code generator. i is the outer loop and j is the inner loop as indicated by the iteration space. We first derive the bounds on j as $0 <= j < i$. We then project j from the system of inequalities to yield the bounds on i as $1 <= i < n$. The final code corresponding to this loop nest is shown in Listing 2.1.

CUDA-CHiLL [39] is a thin wrapper around CHiLL, that utilizes CHiLL's loop transformation capabilities, and finally generates CUDA code once all transformations and code generation steps have been applied, by reducing loops that correspond to parallel CUDA block and thread dimensions and replacing the references to those loops with the correct parallel block and thread indices.

For instance, in Figure 2.3, we see how the tiled loop indices ii and i get replaced as block and thread indices bx and tx, respectively. Subsequently, the loops corresponding to the block and thread indices are removed for the final parallel CUDA code.

2.1.5 Support for nonaffine codes and transformations

The previous sections described iteration spaces, transformations, dependence analysis, and code generation in the polyhedral model with the restriction that all codes and transformations are affine. There has been relatively little research in extending the applicability of the polyhedral model to nonaffine codes or transformations.

```
1 for(i = 1; i < N; i++)
2  for(j = 0; j < i; j++)
3   s0(i,j)
```

Listing 2.1: Polyhedra Scanning.

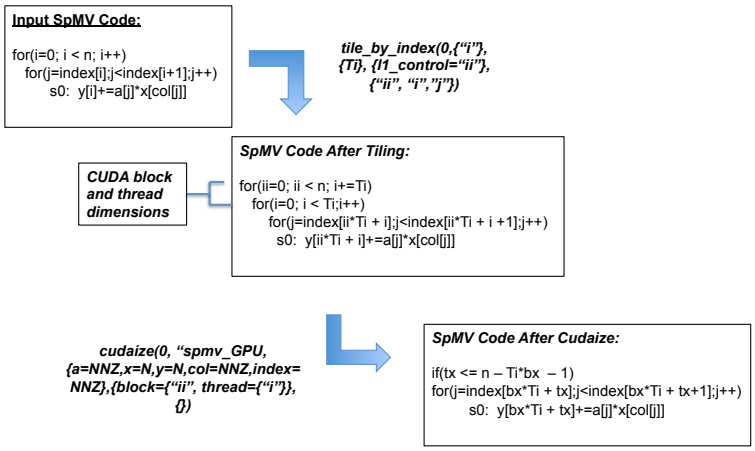


Figure 2.3: CUDA-CHiLL's cudaize transformation.

Analysis and code generation techniques for handling nonaffine loop bounds and array accesses can be broadly grouped into analyses that provide regular representations for irregular codes and run-time inspectors for parallelizing and optimizing such codes.

Compile-time analysis approaches use information about nonaffine accesses to improve the precision of data dependence analysis [40, 41] or introduce conservative approximations in the data dependence analysis [41–45] so that determining the legality of compile-time transformations is possible despite nonaffine loop bounds and/or array accesses. The focus of all these techniques is to tolerate control flow and/or nonaffine memory accesses while still leveraging existing compile-time loop transformations.

Wonnacott and Pugh [40] represented such loop bounds and array accesses with uninterpreted functions. The relationship between the input and output of uninterpreted functions is not known or statically unanalyzable at compile time, hence the term. The Sparse Polyhedral Framework (SPF) [46] also uses uninterpreted functions to represent runtime reordering transformations.

2.2 Representations for Sparse Matrix Vector Multiply(SpMV)

The sparse matrix representations used in this dissertation are depicted in Figure 2.4. In the center, we see a dense representation of matrix A, where over half is comprised of zeros. and detailed below:

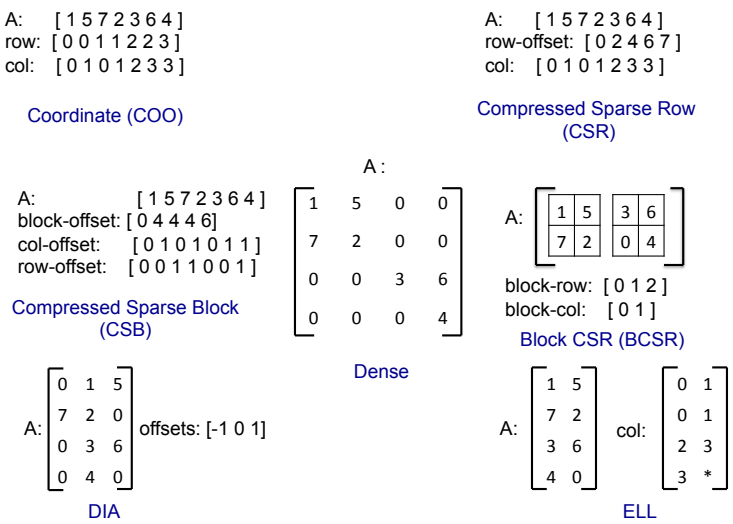


Figure 2.4: Examples of sparse matrix formats

- *Coordinate (COO):* The most general representation uses a vector of just the nonzeros in the matrix, and then 2 auxiliary arrays that represent the column and row for that element.

- *Compressed Sparse Row (CSR):* CSR reduces the space requirements of COO. The column auxiliary array is retained, but the row array has one element per row, indicating the first element for that row.

- *Compressed Sparse Block (CSB):* The CSB format blocks the dense matrix into squares. In Figure 2.4, for example, the dense matrix is blocked into squares of length 2. Each square or block's starting row and column can be uniquely determined and each square's offset into the nonzeros is stored in *block-offset*. The row and column offsets of each nonzero within the square block is stored in 2 auxilliary arrays. CSB saves on indexing memory footprint as the row and column offsets can be stored with potentially less number of bits per index.

- *Block CSR (BCSR):* In BCSR format, the nonzero elements are represented by a collection of small dense blocks, and the blocks are padded where necessary with zero values. An auxiliary array tracks the row and column of the upper left element of each nonzero block.The resulting interior computation can be performed on a small dense array, for which high-performance implementations are more easily obtained.

- *DIA:* The DIA format captures only the diagonals that have nonzero elements. The `offset` auxiliary array represents the offset from the main diagonal. It is well-suited for representing banded matrices.

- *ELL:* The ELL format uses a 2-dimensional matrix with a fixed number of nonzeros per row, and rows with fewer nonzeros are padded with zero values. An auxiliary `col` matrix tracks the columns for the nonzeros as in CSR. When most rows have a similar number of nonzeros, ELL leads to more efficient code because of a fixed number of iterations and no indirection in the loop bounds.

2.3 Inspector/executor methodology

A general technique to analyze data accesses through index arrays and consequently reschedule or reorder data at run time employs an *inspector/executor* paradigm whereby the compiler generates *inspector* code to be executed at runtime that can collect the index expressions and then an *executor* employs specific optimizations that incorporate the run-time information [14, 47–49]. These inspector/executor optimizations have targeted parallelization and communication [8, 48] and data reorganization [9–13].

2.3.1 Iteration and data space reordering

Ding and Kennedy [9] introduced run-time techniques for reorganizing data and the associated computation based on the memory access patterns. Specifically they reorganized the data based on first touch access order in loop based computations, termed consecutive packing or CPACK. Another scheme they introduced was lexicographic grouping, where all iterations that reference the same data are scheduled consecutively. Strout et al. [15] extend these techniques by allowing various different iteration and memory reorganization strategies to be composed with additional inspector-executor strategies such as graph partitioning and full sparse tiling. The ability to compose the specifications for such transformations is implemented. Such transformations are represented using uninterpreted functions. Basumallik and Eigenmann [14] inspect and restructure loops in converting OpenMP programs to Message Passing Interface (MPI) programs. They restructure and reorder iterations to minimize communication of remote data in a distributed computation setting. They employ runtime inspection to accomplish this.

2.3.2 Dependence testing and parallelization

Other approaches such as the the Lazy Reduction-parallelization and Privatization Do-all (LRPD) test [48], speculatively execute a parallel version of the input code at runtime while simultaneously doing a parallel dependence test to check if the parallelization was indeed legal. One extension of the above approach is to aggregate loop iterations into tiles, and do the dependence check across tiles for more coarse grained parallelism, and execute the tiles in a wavefront or DO-ACROSS manner where the inter tile dependences are respected and the code is partially paralllelized.

Another approach to exploiting performance from partially parallel loops is to explicitly construct the full inter-iteration dependence graph in one pass and compute the iterations belonging to each wavefront and then schedule each wavefront in parallel with a barrier or point-to-point synchronization across wavefronts. This is the approach taken to parallelize the forward and backward sparse triangular solver [50–55]. The wavefront computation is done using the SpMP library [56]. We utilize the SpMP library for the above mentioned functionality as well as their point-to-point synchronization primitives to parallelize wavefront style computations.

2.4 Sparse matrix applications

In many cases, the choice of sparse representation to yield high performance tends to be application specific. In this section, we outline the sparse representations and algorithms in a range of sparse matrix applications.

2.4.1 Locally Optimal Block Preconditoned Conjugate Gradient (LOBPCG) and Sparse Matrix-Matrix Multiply (SpMM)

SpMM applies SpMV to multiple right hand sides or vectors. Recent work uses a block eigensolver, LOBPCG, to compute properties of light atomic nuclei accurately in the Many-body Fermion Dynamics for nuclei(MFDn) code [24].

This code spends a substantial amount of time in eigenvalue computations that involve SpMM and SpMM transpose(SpMM_T). Due to the large size of the input matrix, which is symmetric, the Compressed Sparse Block(CSB) sparse matrix format is chosen as an appropriate format for implementation purposes. The CSB format conceptually blocks the columns and rows of the input matrix into square tiles and stores the nonzeros falling in

each tile in the Coordinate(COO) format. This enables the offsets of the nonzeros within a given tile to be stored with 16 bits rather than the full 32 bits to save memory requirements. Further, the CSB format is amenable to parallelization for both SpMM and SpMM_T by either parallelizing across block rows or block columns, respectively [57].

2.4.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent(SGD) is an iterative algorithm over a sparse matrix that computes a low-rank approximation,i.e given an incomplete matrix A, it computes 2 low rank matrices W and U such that A $\approx$ W*U. SGD computes the missing entries of the matrix A. It is used in recommendation systems, for example, to derive a complete set of user-movie ratings from an incomplete database of ratings, where all movies have not been rated by all users [25]. The input ratings are represented as a bipartite graph, with the users and movies corresponding to vertices and the ratings being edges. Each movie and user has a feature vector associated with it and the SGD algorithm sweeps the graph, iteratively updating the feature vectors until they are approximately equal to the edge weight or rating. All originally missing ratings are computed once the algorithm converges.

Recent work [25] uses a variety of online and offline scheduling strategies for SGD. SGD is challenging to parallelize as edges can only be processed in parallel if they do not share any endpoint or vertex; otherwise there is said to be a conflict which will materialize as a race condition at runtime. Online schemes detect and resolve conflicts at runtime for example, by using locks. Offline schemes try to compute conflict free schedules by preprocessing the graph. One offline strategy is to recognize that in the adjacency matrix representation of the graph, processing the entries belonging to the same diagonal is conflict-free, but that synchronization is required across different diagonals. CUDA-CHiLL was used to transform the SGD code based on a CSR representation to one based on the DIA representation and derive high performance GPU code. Additionally, a block-diagonal variant was also developed.

2.4.3 Preconditioned Conjugate Gradient (PCG)

Conjugate Gradient is a popular iterative method for solving a sparse system of linear equations. In this dissertation, we parallelize the conjugate gradient algorithm using either a symmetric Gauss Seidel preconditioner or an Incomplete LU(0) preconditioner. A signif-

icant amount of time in the PCG algorithm is spent in forward and backward Gauss Seidel relaxations. These computations have loop carried dependences, but the outermost loop of Gauss Seidel and Incomplete LU can be partially parallelized using an inspector-executor methodology to derive wavefronts, where iterations within a wavefront may be executed in parallel. However, synchronization is required across wavefronts. As outlined in Section 2.3.2, we compute the inter-iteration dependence graph and then aggregate wavefronts from the dependence graph, using the open source SpMP [56] library, to execute the iterations in parallel.

2.4.4 Verification and validation

Inspector/executor transformations should preserve the data dependences in the original code, that is, they should not reverse the source and sink of any dependence. The general task of partitioning the data dependence relation is undecidable, although it is decidable for affine codes.

For nonaffine codes, inspectors are utilized to inspect nonaffine accesses and the dependences due to them. The inspector reorganizes or partitions the computation subject to the constraints of the dependence relation. Inspectors may also reorganize the code and data for better memory locality. Reordering memory accesses is equivalent to constructing a spatial schedule assignment to the data dependence relation traversal.

Parallel execution of the reorganized code in a wavefront fashion is also constrained by dependences across partitions. If the inspector is unable to partition the data dependence relation then the executor is sequential.

Verifying the correctness of inspector/executor transformations for wavefront parallelization has been studied by Norrish and Strout [58]. Norrish and Strout use an action graph abstraction, constructed at runtime, that captures all ordering constraints between iterations. They then verify the correctness of the inspector/executor transformations by checking that the reordered and/or parallelized computation's action graph is equivalent to the original computation's action graph. These techniques can be combined with our work to prove the correctness of our inspector/executor transformations.

When checking for the correctness of sparse matrix computations where floating point operations are involved, it may not be possible to produce bitwise identical results espe-

cially when the computations are parallelized. Floating point computations are exemplars of interval arithmetic, and all the computations in this dissertation fall under this category. For computations involving interval arithmetic, as long as the computed result encloses the exact unknown result, the inclusion property of interval arithmetic is satisfied [59]. In all our computations, we test for correctness using this property by checking if the computed results match that of the reference sequential implementation, within a certain tolerance that is appropriate for the computation.

2.5 Summary

In this chapter, we presented a brief background on the main components of polyhedral compilation, namely representing iteration spaces, transformations, representing and analyzing dependences, and polyhedra scanning of the final iteration spaces during code generation. We also touched on past research on incorporating nonaffine analysis into compilers. We described the inspector/executor methodology in the context of applying reordering code and data transformations. Finally, we detailed common sparse matrix formats and the optimized representations developed by a compiler and runtime approach as well as the sparse matrix applications optimized in this dissertation.

CHAPTER 3

NONAFFINE EXTENSIONS: REPRESENTATIONS AND TRANSFORMATIONS

Codes with indirect array access expressions, for example, A[B[i]], in loop bounds and/or array subscript expressions are termed nonaffine. Many polyhedral frameworks [28, 29, 60] fail to recognize and/or represent such expressions, let alone optimize them. We utilize the uninterpreted function symbol abstraction from Omega/Omega+ [27, 61] to represent such constructs that arise in the code. Aside from Omega+, the Sparse Polyhedral Framework (SPF) [46] is the only framework that represents index arrays using the uninterpreted function symbols. SPF also uses uninterpreted functions for Run-Time Reordering Transformations (RTRTs) [38].

In this chapter, we detail our support for both representing nonaffine code constructs and transforming codes using nonaffine transformations. We demonstrate how representing nonaffine loop bounds using the uninterpreted function abstraction allows us to transform those loops. Next, we introduce the *generalized* loop coalescing transformation that converts a multidimensional loop nest, potentially containing nonaffine loop bounds, into a single dimensional loop. For this transformations, we utilize the same uninterpreted function abstraction for encapsulating run-time inspector functionality in the form of nonaffine mappings. With these extensions, we can compose with existing polyhedral transformations via unified abstractions for affine and nonaffine mappings.

3.1 Nonaffine representations: Loop bounds and subscripts

The code in Listing 3.1 shows a typical serial implementation of SpMV. The inner `j` loop has nonaffine loop bounds and the reference `x[col[j]]` has a nonaffine array subscript. Most polyhedral compilers will stop at just analyzing the outer `i` loop alone, since it has affine loop bounds. However representing the `j` loop has certain advantages for

```
1 for(i = 0; i < N; i++)
2   for(j = index[i]; j < index[i+1]; j++)
3         y[i] += A[j]*x[col[j]];
```

Listing 3.1: SpMV Code based on the CSR format

parallelization as will be shown in Chapter 4. The iteration space for the SpMV code above is represented in our system as follows:

$$I = \{[i,j] \mid 0 \leq i < N \ \wedge \ index(i) \leq j < index(i+1)\} \tag{3.1}$$

The compiler parses the code shown in Listing 3.1 and recognizes an array expression in the loop bounds of `j`. It encodes `index` as an uninterpreted function of the outer loop iterator i with arity 1, that is a single argument. It also encodes the argument to the uninterpreted function as a relation. The inputs to the relation are the outer loop variables and the output for the relation is the array subscript expression. For instance, the upper bound on the j loop is index[i+1], and the subscript expression, i+1 maybe represented as the affine function i $\rightarrow$ i+1 by the compiler. This has been termed an array access function in prior work [38].

Throughout the transformation process the uninterpreted functions are treated in a fashion identical to global variables, where global variables represent read-only parameters. For correctness, these indexed loop bounds must be functions of iterators in the enclosing loops.

The representation of nonaffine loop bounds utilizing uninterpreted functions paves the way for transformations or mappings to manipulate these loop bounds. For example, the tiling transformation might be effected on the above iteration space using the following relation T:

$$T = \{[i,j] \rightarrow [i,jj,j] \mid (\exists \alpha : jj = 4\alpha \ \wedge \ jj \leq j < jj+4)\} \tag{3.2}$$

to result in the following tiled iteration space:

$$I' = \{[i,jj,j] \quad | \quad (\exists \alpha : jj = 4\alpha \tag{3.3}$$

$$\wedge j - 4 < \ jj \leq j < index(i+1) \tag{3.4}$$

$$\wedge \ 0 \leq i < N \ \wedge \ index(i) \leq j)\} \tag{3.5}$$

α is an existential used to represent the tile size and possible values for the tile controlling loop's starting point.

3.1.1 Redefinition of statement interface for nonaffine transformations

Strout et al. [15, 38] used uninterpreted functions to represent transformations with an inspector and executor. To incorporate inspector-based transformations into existing polyhedral compiler frameworks, the statement macro interface discussed in Chapter 2 must change.

This section presents a modification to the statement macro interface that enables the simplification of array access functions needed to incorporate inspector-based transformations. As an example assume the following loop:

```
for (i=1; i<N; i++) {
    ... X[ i-1 ] ... Y[ f[ i ] ] ...
}
```

The following relation T represents a permutation transformation of the above loop, where f is an uninterpreted function at compile time.

$$T = \{[i] \rightarrow [j] \mid j = f(i) \wedge 1 \leq f(i) < N\}$$

One way to derive the values of f is to use an inspector at runtime. When performing transformations it is necessary to solve for the old iterator in terms of the new iterator so that any array accesses in the loop now uses the new iterator. This is equivalent to composing the array access relation with the inverse of the transformation mapping. In the context of the above example, the array access relation for X would be $i \rightarrow i - 1$ and for Y would be $i \rightarrow f(i)$. Since the transformation is represented by the permutation f the inverse of the transformation is f^{-1}. The new subscripts would be obtained by composing each array's access relation with the inverse of the transformation.

When we replace the old iterator i with a function of the new iterator j, the new access expressions become `X[ f_inv[ j ] - 1 ]` and `Y[ f[f_inv[ j ] ] ]` , as shown in Figure 3.1(a). The compilation framework needs to maintain the knowledge that f is a bijection, and therefore invertible, and that the inverse `f_inv` is provided by an inspector.

a. Statement macro interface and usage before

```
#define S0_before(i) ... X[(i)-1] ... Y[f[(i)]] ...
for (j=1; j<N; j++)
  S0_before( f_inv[ j ] );

// Code after preprocessing
for (j=1; j<N; j++)
  ... X[f_inv[j]-1] ... Y[f[f_inv[j]]] ...
```

b. Statement macro interface and usage after

```
#define S0_after(r0,r1) ... X[r0] ... Y[r1] ...
for (j=1; j<N; j++)
  S0_after( f_inv[j]-1, j );

// Code after preprocessing
for (j=1; j<N; j++)
 ... X[f_inv[j]-1] ... Y[j] ...
```

Figure 3.1: Current code generators pass to statement macros old iterators as a function of new iterators. To incorporate the results of inspector-based transformations and enable simplification of access expressions, we modify the statement macro interface so that whole access expressions are passed to the statement macro.

An explanation of how composition and applying a transformation relation to a set are performed when uninterpreted functions are involved is provided in [46].

Code generators that use the typical statement macro approach would pass `f_inv[ j ]` into the statement macro (see Figure 3.1(a)). A backend compiler like gcc cannot do the simplification that uses the equality $f(f^{-1}(j)) = j$, so two unnecessary levels of indirect referencing stay in the code.

We solve this problem by modifying the statement macro interface in a simple but important way. Now instead of having a parameter in the statement macro for each iterator in the original loop, there is a parameter in the statement macro for each memory/array access in the original statement (see Figure 3.1(b)). For example `Y[ f[i] ]` will become `Y[ r1 ]` in the statement macro. Then the code generator when generating the call to the statement macro provides the modified access function, which in this example will result in the access expression for `Y` being `j` instead of `f[f_inv[j]]`. The PPCG compiler [22] also represents access relations and simplifies them before generating code. It uses the representation of access relations to map data to registers and allocating shared memory storage for tiles in generated CUDA code. However, the PPCG compiler does not handle nonaffine access relations.

The main impact of this change for the compiler is that the representation of the computation needs to include a relation for each array access expression for each statement, in addition to the statement's iteration space specification.

The above approach exploits the polyhedral representation to chain transformations, where each iteration space transformation is composed with array access relations at com-

pile time. It also has the added advantage that the mapping and iteration space abstractions can be manipulated or simplified to result in high performance code.

3.2 Nonaffine transformation: Generalized loop coalescing

Suppose we want to parallelize a COO representation of SpMV, starting from the CSR representation of Listing 3.1, this section shows how the code generation is extended and integrated with a compiler-generated inspector to perform a run-time transformation we call *generalized loop coalescing*.The inspector/executor code generation for the generalized loop coalescing transformation is outlined in Figure 3.2. This transformation combines the inner and outer loops into a single loop with an expanded iteration space. In the vectorization literature, coalescing is applied to loop nests with affine bounds to generate longer vectors, and the subscript expressions are effectively linearized to map to the coalesced loop's bounds [62]. Our generalization to nonaffine loop bounds is similar to that of Basumallik and Eigenmann[14], but their approach is exclusively syntax-based and not integrated into a polyhedral transformation framework. We implement the coalescing transformation as a loop transformation, representing it as an uninterpreted function along

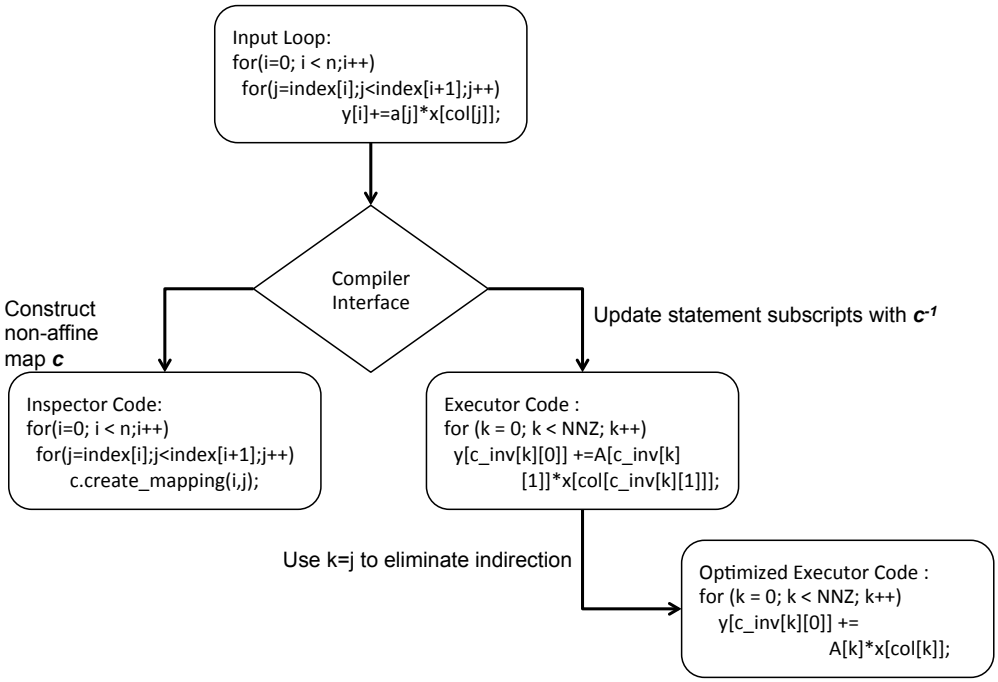


Figure 3.2: Overview of the generalized loop coalescing transformation. The steps involving both inspector and executor code generation are shown, including optimizations for executor code, based on additional simplification.

with syntax components.

To determine the iteration space, we must generate code for an inspector, which creates a mapping from the original iteration space of the inner and outer loops to that of the coalesced loop. Once this enabling transformation has been applied, other transformations on the iteration space become valid, and the polyhedral framework can be used.

3.2.1 Coalescing transformation relation

Generalized loop coalescing is a nonaffine transformation that converts the input loop nest's iteration space with multiple dimensions to one with a single dimension. As compared to the affine shifting transformation in Chapter 2, where the inverse of the transformation *xform_inv* is a bijective function that exactly specifies the relationship between the input and output loop indices of the transformation, the relationship between the input iteration space and the output iteration space is not known at compile time and is only fully realized at run time. We introduce the following relation to represent this mapping, using an uninterpreted function c to represent the mapping from the original loop iterators to the coalesced loop iterator derived from an inspector. The coalesced loop's upper bound is represented by the variable, `NNZ`, and is determined by the inspector at runtime.The following relations show the 2-D to 1-D mapping, and the n-D to 1-D mapping, respectively.

$$\begin{aligned} T_{coalesce} &= \{[i,j] \rightarrow [k] \mid k = c(i,j) \wedge 0 \leq k < NNZ\} \\ \{[I_1, I_2, I_3, \ldots, I_n] &\rightarrow [k] \mid k = c(I_1, I_2, I_3, \ldots, I_n) \wedge 0 \leq k < NNZ\} \end{aligned} \tag{3.6}$$

Our framework uses the information that the coalescing function c is bijective and that the inspector makes the inverse of c available, c^{-1}.

3.2.2 Inspector for coalescing

The loop coalescing transformation establishes the values of the uninterpreted function c with compiler-generated runtime *inspector code*. A sample code snippet corresponding to the inspector is shown in Figure 3.3. The call to the function `create_mapping(i,j)` sets up the mapping corresponding to the uninterpreted function c, from which the inverse mapping c^{-1} can be derived. The loop coalescing transformation creates a C++ class as an abstraction for the functionality of the uninterpreted function. It creates one array per

```
struct access_relation {

  // array to track old iterators
  int c_inv[][2];

  // variable to keep track of k
  int k;
  void create_mapping(int i,int j){
      c_inv[k][0] = i;
      c_inv[k][1] = j;
      k++;
   }
}
```

```
struct access_relation c;
for (i=0; i<=n-1; i++)
 for (j=index[i]; j<=index[i+1]-1; j++)
    c.create_mapping(i,j);
```

Figure 3.3: Compiler-generated coalescing inspector.

index variable being coalesced local to the uninterpreted function data structure. It also has a scalar variable to indicate the range of the uninterpreted function. This step is similar to the creation of the inspector in [14], where the inspection is done on a per index basis and the values of the output variable corresponding to the each index is recorded.

To construct the mapping from the old loop nest to the new one, the original loop nest has to be traversed once by the inspector using the same iteration space as the original nest. Hence, the loop coalescing transformation creates a new statement for the inspector routine and copies the old iteration space of the untransformed statement to that of the inspector's iteration space.

Since the inspector code is sequential, its associated overhead has to be amortized over many calls to SpMV. However this is not uncommon, as in the case of the Conjugate Gradient solver that calls the SpMV kernel multiple times over which the input matrix structure remains invariant.

3.2.3 Code generation

The next task of the coalescing transformation is to set up the iteration space of the transformed statement and update the changes in the indexing expressions of the transformed code (sometimes called the *executor*). Conceptually, the compiler has to remove the original indices, and replace them with uninterpreted function inverses.

After applying the loop coalescing transformation $T_{coalesce}$ on the CSR SpMV code in Listing 3.1, the old iterators i and j in terms of the new iterator k are as follows: $i =$

$c^{-1}(k)[0]$ and $j = c^{-1}(k)[1]$, where c^{-1} is a function that returns a 2-tuple. The resulting code becomes:

```
for (k = 0; k < NNZ; k++)
  y[c_inv[k][0]]
    += A[c_inv[k][1]]*x[col[c_inv[k][1]]];
```

The process of simplifying access expressions outlined in Section 3.1.1 did not result in improved code in this case. However, it is possible to use program analysis techniques such as those presented by Lin and Padua [41] on the inspector to determine that $j = k$. Using the constraint $j = k$ in the loop coalescing transformation

$$T'_{coalesce} = \{[i,j] \rightarrow [k] \mid j = k \wedge k = c(i,j) \wedge 0 \leq k < NNZ\} \tag{3.7}$$

enables simplification of access expressions so that the resulting code is instead

```
for (k = 0; k < NNZ; k++)
  y[c_inv[k][0]] += A[k]*x[col[k]];
```

In our experimental results, we show the performance due to the original loop coalescing transformation with and without the $j = k$ constraint. Utilizing this additional constraint is important to reduce the number of indirect accesses in the resulting code.

3.3 Summary

This chapter discussed the compiler extensions for nonaffine loop bounds and transformations which utilize the uninterpreted function symbol abstraction. Representing nonaffine loop bounds allows us to compose further transformations, such as tiling. The statement macro interface for polyhedral code generation was described. Additionally, the modifications for nonaffine transformations was discussed. Specifically simplification of the array subscript expressions to minimize indirection when subjected to nonaffine mappings was explained.

Generalized loop coalescing was introduced as a nonaffine transformation that converts a loop of multiple dimensions, potentially containing nonaffine loop bounds, to a single loop. It uses an uninterpreted function to record the nonaffine mapping between the input and output loop iterators. The executor code is updated with the inverse of

the uninterpreted function at compile-time while the inspector code explicitly constructs this mapping at runtime. We also discussed compile-time optimizations to reduce the indirection in the executor code resulting from generalized loop coalescing.

CHAPTER 4

PARALLELIZATION OF SPMV UTILIZING NONAFFINE EXTENSIONS

In this chapter, we show how the nonaffine representations and transformations from the previous chapter can be composed with other affine transformations, specifically tiling, fusion, distribution, peeling, and scalar replacement and expansion [26]. Additionally our framework supports an *array reduction* that interfaces with Nonaffine indices. The array reduction implementation is integrated with a highly tuned GPU specific implementation from Nvidia's CUSP [5] library. We demonstrate the application of our framework with the aforementioned components in generating high performance SpMV code for the CSR and COO matrix representations. We also support different parallelization strategies, such as parallelizing across nonzeros or rows and nonzeros within a row. With these extensions our framework achieves performance that is comparable and sometimes outperforming the manually tuned CUSP library on the Nvidia Tesla C2050 Fermi GPU.

4.1 GPU optimization considerations

To justify the GPU-specific transformations described in this section, we first clarify features of the GPU architecture that influence the selection of the implemented approach.

Critical to the parallelization strategy is the two-level parallelism hierarchy of the the architecture. Several SIMD processors (SP) form a streaming multiprocessor (SM). The GPU consists of several SMs that operate independently with their own control (MIMD). CUDA blocks are indivisibly mapped to SMs, and CUDA threads are mapped to SPs. Within a block, a warp is the set of contiguous SIMD threads that are scheduled together, and can be exploited to reduce synchronization. This organization impacts the overhead of synchronization: synchronization within a warp is not needed, within a block it has low overhead due to the SIMD single thread of control, but across blocks it is very expensive. The resulting reduction implementation exploits these different levels of synchronization

overhead to lead to its efficient implementation.

We must also consider how the memory system organization affects the reduction implementation. The GPU consists of a single device DRAM, called *global memory*, which is addressable and shared by all blocks. As with any off-chip DRAM, global memory accesses observe high latencies in the hundreds of cycles, so optimizing global memory accesses can significantly impact performance. The bandwidth is maximized when global memory accesses across threads can be combined into a single memory transfer, called *global memory coalescing*. Coalescing is achieved when accesses across threads in the same warp appear in the same contiguous transfer unit; this typically requires that adjacent threads access adjacent data. Each block also has a scratchpad memory that is shared across its threads, but inaccessible to other blocks; this storage is called *shared memory*, Shared memory has extremely low latency, but its capacity is severely limited.

4.2 Reduction transformation

The reduction transformation that interfaces with the Nonaffine indices is a parallel reduction. As is common in parallelizing compilers, high-performance code that implements a reduction is architecture specific, and so the code generation strategy is to transform the code and then invoke a highly-tuned architecture-specific library to implement the accumulation into a final result. We have developed a set of reduction functions that are implemented with two levels of parallelism corresponding to the block and thread hierarchy in GPUs. These functions are similar to those used by CUSP, but are designed to interface with a compiler. This section describes the reduction library functions, how they interface with the index expressions from the inspector, and requirements for code transformations to set up the code for these library calls.

4.2.1 Identifying reductions

The original double loop corresponding to CSR SpMV, shown in Listing 3.1 has a flow, anti and output dependence on y carried by the inner loop due to the update, prior to coalescing. However, the only update to vector y is a commutative and associative operation, which can be recognized as a reduction, as is common among parallelizing compilers [63, 64]. In our compiler, we mark the dependences on y as being a reduction,

to make it possible to parallelize either loop level. Additional information is recorded in case reduction is combined with loop coalescing: the outer loop bounds do not have any indirection, are monotonic and have a unit stride.

4.2.2 Reduction library

Figure 4.1 illustrates a general two-level reduction that can be used to exploit the two-level parallelism hierarchy of the GPU (threads and blocks). The reduction shown here is a *segmented reduction*, where matrix rows may span across warp and/or block boundaries. A sequence of adjacent input values (corresponding to rows in the original matrix) are accumulated into a single output value (corresponding to a single element of y). This transformation is used for the COO format implementation in the next section, and is a generalization of what is used for the CSR Vector implementation.

In the first step of the reduction each thread copies a corresponding entry from the input matrix and the input vector from global memory, computes the product and stores the result in shared memory. A segmented reduction is then done in shared memory. Each segment is denoted with a particular color in Figure 4.1.

A collection of threads that are executed in a single instruction issue on a GPU are called a *warp*, shown as 8 threads in Figure 4.1. Whenever the last thread of the row does not fall on a warp boundary, the thread adds its accumulated sum to the output vector in

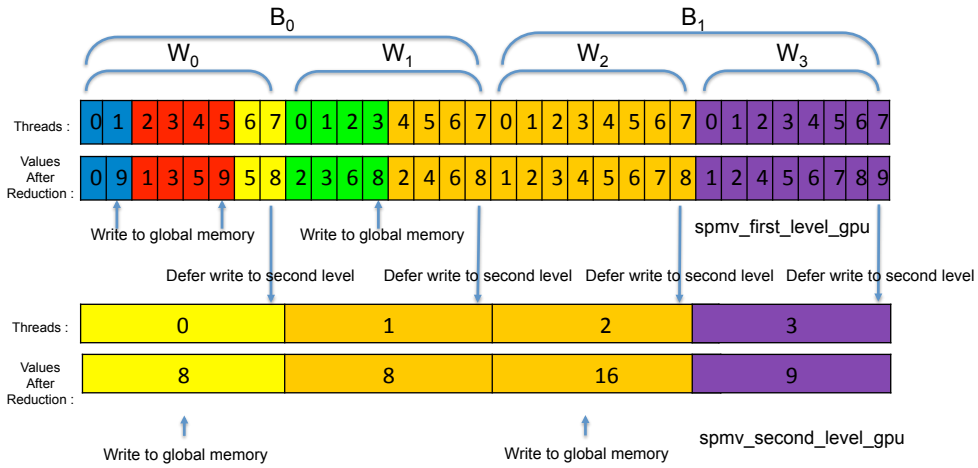


Figure 4.1: GPU implementation of 2-level parallel reduction over a block/thread-warp hierarchy. $B_{0,1}$ represent GPU blocks and W_{0-3} represent thread-warps.

global memory, for instance threads 1 and 5 within Warp 0 in Figure 4.1. Otherwise, the last thread of each warp defers its write to global memory by writing to an intermediate array stored in the GPU block's shared memory that is reduced in the second phase. The 2-level reduction conceptually updates row-sums spanning across warp block boundaries. For CSR vector, since each warp processes exactly one row, the 2nd level reduction proves unnecessary, and the last thread of the warp updates the row sum at the end of the first level reduction.

The reduction for the CSR Vector format is simpler than the COO format reduction shown in Figure 4.1, as rows do not cross warp boundaries. For CSR vector, the 2nd level reduction proves unnecessary, and the last thread of the warp updates the row sum at the end of the 1st level reduction.

4.3 Transformations for GPUs

We now describe how this system was used to generate three versions of GPU code achieving high performance for SpMV, following the versions in [5]. The three versions differ in parallelization strategy: *CSR Scalar* parallelizes only across rows, *CSR Vector* parallelizes across rows as well as the reduction across nonzeros within a row and *COO* parallelizes across nonzeros disregarding row boundaries. It utilizes a segmented reduction with the row identifier corresponding to the segment. These versions and their parallelization are ilustrated in Figure 4.2 using a simple sparse matrix with 10 nonzeros. As is evident from the figure, *CSR Scalar* assigns one thread per row processed, while *CSR Vector* assigns a group of threads per row and *COO* assigns a group of threads per batch of nonzeros. Here the group size is 4 for purposes of illustration, but in actuality the group size is the warp size which is 32.

For each version, we used the compiler framework described in the previous section. The input to the compiler was the original sequential SpMV code from Chapter 3, and a CUDA-CHiLL script that describes the transformations to be applied. The transformation sequences address the GPU specific optimization considerations outlined in Section 4.1.

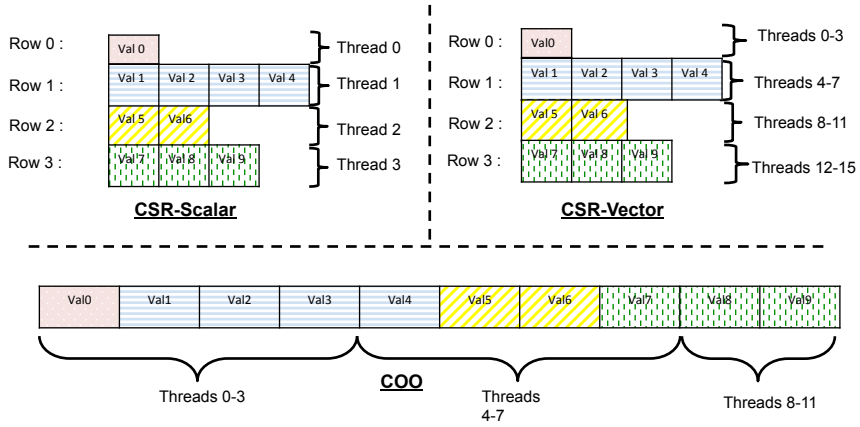


Figure 4.2: Different parallelization strategies for SpMV.

4.3.1 Parallelizing rows: CSR Scalar

Of the three SpMV implementations, CSR Scalar code is the simplest to generate, so it is only discussed briefly. The CUDA-CHiLL script and compiler-generated output are shown in Figure 4.3(a) and (b). The dot product across each row of the input matrix is processed in parallel, with each thread being assigned a single row and computing its elements sequentially. Loop i corresponding to the matrix row is tiled. In the `cudaize` command, the tile controlling loop is mapped to the blocks, and the tile is mapped to the threads in the GPU's two-level parallelism hierarchy.

CSR Scalar suffers from load imbalance across threads especially when the number of nonzeros per row exhibit high variance. Further, accesses to global memory are not coalesced as consecutive threads are not accessing consecutive elements in global memory, resulting in significant memory traffic.

4.3.2 Parallelizing within a row: CSR Vector

For CSR Vector, the CUDA-CHiLL script and compiler-generated output are shown in Figure 4.3(c) and (d). Two levels of tiling are used, across rows and within a row for the intra-warp reduction. The second level of tiling for the intra-warp reduction is done on the `j` loop with index arrays in the loop bounds, and is only possible because of the Nonaffine representation of the index arrays involved in the loop bounds utilizing uninterpreted functions as outlined in Chapter 3.

The `cudaize` command maps the resulting loops to the 1-D block and 2-D thread di-

a. CSR Scalar Script

```
tile_by_index(0,{"i"},{Ti},{l1_control="ii"},
{"ii","i","j"})CU=1

cudaize(0,"spmv_GPU",{ a=NNZ,x=N,y=N,col=NNZ,
index=NNZ}, {block={"ii"}, thread={"i"}},{})
```

b. CSR Scalar Code

```
__global__ void spmv_GPU(float *y,float *a,float *x,int *col,int *index) {
   …
if (tx <= NROWS – TILESZ* bx - 1)
  for (j=index_(TILESZ* bx + tx);j<=index__(TILESZ*bx + tx)-1;j+=1)
    y[TILESZ* bx + tx] += (a[j] * x[col[j]]);
}
```

c. CSR Vector Script

```
tile_by_index(0,{"i"},{Ti},
 {l1_control="ii"},{"ii","i","j"})CU=1
tile_by_index(0,{"j"},{Tj},{l1_control="jj",l1_tile="j"},
{"ii","i","j", "jj"},strided)CU=1

scalar_expand_by_index(0,{"i","j"},"RHS",
CP_TO_SHARED,
NO_PAD,ACCUMULATE_THEN_ASSIGN)

cudaize(0,"spmv_GPU",{ a=NNZ,x=N,y=N,
col=NNZ,index=NNZ},{block={"ii"}, thread={"j", "i"}},{})

reduce_by_index(0,{"jj"}, "reduce_warp",{}, {"tx"})
```

d. CSR Vector Code

```
#define index_(i) index[i]
#define index__(i) index[i + 1]
__global__ void spmv_GPU(float *y,float *a,float *x,int *col,int *index) {
  …
__device__ __shared__ float _P1[TILESZ*WARPSZ];
 if (ty <= NROWS - TILESZ* bx - 1) {
   if (tx <= index__(ty + TILESZ* bx) - index_(ty + TILESZ* bx) - 1)
     _P1[tx + ty * WARPSZ] = 0;
   if (tx <= index__(ty + TILESZ* bx) - index_(ty + TILESZ* bx) - 1)  {
     for (jj = index_(ty + TILESZ* bx); jj <= -tx + index__(ty + TILESZ* bx) - 1; jj +=
WARPSZ)
       _P1[tx + ty * WARPSZ] += (a[tx + jj] * x[col[tx + jj]]);

     reduce_warp(&y[ty + TILESZ* bx],&_P1[tx + ty * WARPSZ], _lt(31,index__(ty +
TILESZ* bx) - index_(ty + TILESZ* bx) - 1));
} } }
```

e. COO Script

1. Coalescing

```
kernel1 = coalesce_by_index(0,"cidx",{"i","j"}, "c")
kernel3 = split_with_alignment_by_index(kernel1,"cidx",Ti)
distribute_by_index({kernel1,kernel3},"i")
```

2. Tiling

```
tile_by_index(kernel1,{"cidx"},{Ti},{l1_control="block"},
{"i","j","block","cidx"})
tile_by_index(kernel1,{"cidx"},{Tj},{l1_control="warp"},{"i", "j",
"block","warp","cidx"})
tile_by_index(kernel1,{"cidx"},{Tk},{l1_control="by_warp"},{"i", "j",
"block", "warp","by_warp","cidx"})
```

3. Setup for segmented 2-level reduction

```
kernel2 = setup_for_segreduce(kernel1,"warp",{"block", "warp",
"cidx"}, segment,PADSZ,shm,TILESZ_LEVEL2,"k", stmt_to_reduce)
```

4. Setup for segmented last-level reduction

```
tile_by_index(kernel3,{"cidx"},{TILESZ_LEVEL3},
{l1_control="block"},{"i", "j", "block",  "cidx"},strided)
scalar_expand_by_index(kernel3,{"cidx"},segment,shm, NO_PAD,
ASSIGN_THEN_ACCUMULATE)
scalar_expand_by_index(kernel3,{"cidx"},"RHS",shm, NO_PAD,
ASSIGN_THEN_ACCUMULATE)
```

5. Cudaize

```
cudaize(kernel1,"spmv_first_level_gpu",
{ a=NNZ,x=N,y=N,col=NNZ, temp=NNZ, c_j=NNZ, c_i=NNZ },
{block={"block"}, thread={"cidx", "warp"}},{"_P_DATA1",
"_P_DATA2"})
cudaize(kernel2,"spmv_second_level_GPU",
{ a=NNZ,x=N,y=N,col=NNZ, temp=NNZ,c_i=NNZ},{block={},
thread={"warp","block"}}, {"_P_DATA1", "_P_DATA2"})
cudaize(kernel3,"spmv_final_level_GPU",
{ a=NNZ,x=N,y=N,col=NNZ, temp=NNZ, c_j=NNZ, c_i=NNZ},
{block={}, thread={"cidx"}}, {})
```

6. Reduce

```
reduce_by_index(kernel1,{"tx"},"segreduce_warp",{"by_warp"}, {})
reduce_by_index(kernel2,{"ty","tx"},"segreduce_block",{},{"ty"})
reduce_by_index(kernel3,{"tx"},"segreduce_block2",{},{"tx"})
reduce_by_index(stmt_to_reduce[1],{"tx"},"segreduce_warp",{},{})
```

f. COO Code

```
__global__ void spmv_first_level_gpu(int *c_count,float *y,int *_P_DATA1,float
*_P_DATA2,int *c_i,float *a,int *c_j,float *x,int *col) {
 …

__shared__ int _P1[4 * 64];
__shared__ float _P2[4 * 64];

for (by_warp = 0; by_warp < DIM_X - 1; by_warp += 1) {
   _P1[tx + ty * 64] = c_i[32 * by_warp + NZB * bx +   NZW* ty + tx];
   _P2[tx + ty * 64] = (a[32 * by_warp + NZB * bx + NZW* ty + tx] * x[col[32 * by_warp
+ NZB * bx + NZW* ty+ tx]]);
   segreduce_warp(&y[0],&_P1[0 + ty * 64],&_P2[0 + ty * 64],by_warp);
 }
 _P1[tx + ty * 64] = c_i[32 *(DIM_X -1) + NZB * bx + NZW* ty+ tx];
 _P2[tx + ty * 64] = (a[32 * (DIM_X -1) + NZB * bx + NZW* ty+ tx] * x[col[32 * (DIM_X
-1) + NZB * bx + NZW*ty + tx]]);
 segreduce_warp(&y[0],&_P1[0 + ty * 64],&_P2[0 + ty * 64]);
 _P_DATA1[ty + bx * 4] = _P1[31 + ty * 64];
 _P_DATA2[ty + bx * 4] = _P2[31 + ty * 64];
}

__global__ void spmv_second_level_gpu(int *c_count,float *y,int
*_P_DATA1,float *_P_DATA2) {
…
__shared__ int _P3[DIM_Y2*DIM_X2];
__shared__ float _P4[DIM_Y2*DIM_X2];

 for (k = 0; k <= (c(t2,t4) - NZB ) /(DIM_Y2*NZB ); k += 1)
   if (ty <= (c(t2,t4) - (DIM_Y2*NZB ) * k - NZB ) / NZB ) {
     _P3[tx + ty*DIM_X2] = _P_DATA1[tx + (DIM_Y2 * k + ty) *DIM_X2];
     _P4[tx + ty*DIM_X2] = _P_DATA2[tx + (DIM_Y2 * k + ty) *DIM_X2];
     segreduce_block(&y[0],&_P3[0 + 0*DIM_X2],&_P4[0 + 0*DIM_X2], _lt(NZW-1,(-
(NZW* NZB *k) + c(1,1) – NZB) / NZB ));
   }
}

__global__ void spmv_final_level_GPU(int *c_count,float *y,int *c_i,float *a,int
*c_j,float *x,int *col) {
 …
__shared__ int _P1[BLOCKDIM2];
__shared__ float _P2[BLOCKDIM2];

 for (cidx= (c(t2,t4)/NZB )*NZB ; cidx<= c(t2,t4) - 1; cidx+= BLOCKDIM2)
   if(tx + cidx<=  c(t2,t4) - 1){
     _P1[tx] =  c_i[cidx+tx];
     _P2[tx] =  a[cidx+tx]*x[col[cidx+tx]];
     segreduce_block2(&y[0],&_P1[0],&_P2[0],_lt(c(t2,t4) - cidx- 1,BLOCKDIM2 -1));
    };
}
```

Figure 4.3: CUDA-CHiLL scripts and the corresponding generated codes for (a)-(b) CSR Scalar, (c)-(d) CSR Vector and (e)-(f) COO.

mensions. To prepare for the intra-warp reduction, each thread computes its corresponding product expression (A*x) and stores it in shared memory. To effect this strategy, the product expression, denoted by the special `RHS` argument, is replaced by a scalar and then expanded across parallel loop levels, (*i* and *j*) using `scalar_expand_by_index`. The arguments `CP_TO_SHARED` and `NO_PAD` indicate that the array created by scalar expansion is to be stored in GPU shared memory, and that the array is not to be padded, respectively. `ACCUMULATE_THEN_ASSIGN` specifies that the array created by scalar expansion is to be accumulated into, so that it has a "+=" sign when it is on the left hand side of the assignment operator, in an compound assignment statement, and the final result is obtained from the array by a simple "=" assignment statement when it is on the right hand side of the assignment operator. This option only has effect when the original statement is a compound assignment statement. Correspondingly, `ASSIGN_THEN_ACCUMULATE` assigns the product expression to the array created initially and then does an accumulation for the final result.

The loop, *jj*, is replaced by a call to a reduction routine (`reduce_warp` in the command, `reduce_by_index`). The CUDA-CHiLL interface makes it possible to replace reductions with any library implementation.

CSR Vector assigns multiple threads to process a row's dot product, and the row sum is computed using a reduction across threads. CSR Vector exhibits better global memory coalescing than CSR Scalar as the multiple threads assigned to a particular row access consecutive elements in global memory. However, its performance can suffer in the face of high variance of nonzeros across rows.

4.3.3 Parallelizing across elements: COO

The COO implementation is by far the most complex, using all the system capabilities described in Chapter 3 and Section 4.2 of the paper. The scripts and corresponding generated code are shown in Figure 4.3(e) and (f), respectively, and we describe the roles of the six boxes in the script.

1. Coalescing. From the initial representation of SpMV for the CSR format, generalized loop coalescing and generation of its supporting inspector routine are implemented by the `coalesce_by_index` command as outlined In Chapter 3. To generate high-performance

steady-state code, iterations that are not divisible by the block size are split off using the remaining commands in this box, using the `split_with_alignment_by_index` transformation. The loops resulting from split are distributed so that they are separate kernels.
2. Tiling. The coalesced loop is then tiled thrice. Two of the 3 loops thus derived correspond to the parallel CUDA block and thread dimensions. The third loop, denoted by the loop index `by_warp` corresponds to the number of times a warp of 32 threads should iterate to process all nonzeros assigned to it. For instance, if each warp is assigned 256 nonzeros to process, then `by_warp` would have a loop count of 256/32 which is 8, since each warp processes a batch of 32 nonzeros at a time.
3. Setup for segmented 2-level reduction. The command `setup_for_segreduce` abstracts the compiler transformations that are needed to set up for the segmented reduction. It firstly peels the last thread of any parallel dimension/loop, which has to be treated differently with regards to committing its write, and it has to save its results in an intermediate array for a second level reduction. It performs scalar expansion to effect the write to an intermediate array in shared memory.

Further, loop distribution separates the peeled statements corresponding to the 2nd level reduction from the 1st level reduction kernel. The steps in the `setup_for_segreduce` transformation are detailed in Listings 4.1 to 4.3. *4. Setup for segmented last-level reduction.* The tile and scalar expansion sequence are for the last level kernel.
5. and 6. Cudaize and Reduce. Finally a sequence of cudaize and reduce commands are shown to map the designated loops to block and thread dimensions and to replace certain loops with the specified reduction routines. The "block", "warp" and "cidx" loops are examples of parallel loops that are mapped to parallel block and thread dimensions in CUDA, and cudaize marks them to be removed and replaced with parallel thread indices during code generation. The "cidx" loop is marked for reduction and is replaced with a call to *segreduce_warp* similar to the way reduction is implemented for CSR Vector.

Since the inspector used by coalescing keeps track of the original indices and their correspondence with the new loop levels, we can preserve the values corresponding to the segment within our framework and supply it to the reduction routines subsequently.

```
/*1.  After tiling, NZB = nonzeros per block , NZW = nonzeros per warp
    DIM_X =  NZW/32,  index = cidx + by_warp*32 + warp*NZW + block*NZB;
*/
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++)
   for(by_warp=0; by_warp < DIM_X; by_warp++)
    for(cidx=0; cidx < 32; cidx++)
     y[c_i[index]] += A[index]*x[col[index]];
/*2. Scalar expand across indices (warp,cidx) for A*x and c_i*/
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++)
   for(by_warp=0; by_warp < DIM_X; by_warp++)
    for(cidx=0; cidx < 32; cidx++){
     _P1[warp][cidx] = c_i[index];
     _P2[warp][cidx] =  A[index]*x[col[index]];
     y[_P1[warp][cidx]] += _P2[warp][cidx];}
/*3.Peel last iteration of by_warp loop since the last thread in a warp
    cannot commit its write, has to defer to second level kernel*/
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++){
   for(by_warp=0; by_warp < DIM_X - 1; by_warp++){
    for(cidx=0; cidx < 32; cidx++){
     _P1[warp][cidx] = c_i[index];
     _P2[warp][cidx] =  A[index]*x[col[index]];
     y[_P1[warp][cidx]] += _P2[warp][cidx];}}
   for(cidx=0; cidx < 32; cidx++){
    _P1[warp][cidx] = c_i[index];
    _P2[warp][cidx] =  A[index]*x[col[index]];
    y[_P1[warp][cidx]] += _P2[warp][cidx];}}
/*4.Peel the 31st thread of the newly introduced loop for the write to y
    */
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++){
   for(by_warp=0; by_warp < DIM_X - 1; by_warp++){
    for(cidx=0; cidx < 32; cidx++){
     _P1[warp][cidx] = c_i[index];
     _P2[warp][cidx] =  A[index]*x[col[index]];
     y[_P1[warp][cidx]] += _P2[warp][cidx];}}
   for(cidx=0; cidx < 31; cidx++){
    _P1[warp][cidx] = c_i[index];
    _P2[warp][cidx] =  A[index]*x[col[index]];
    y[_P1[warp][cidx]] += _P2[warp][cidx];}
   _P1[warp][31] = c_i[index];
   _P2[warp][31] =  A[index]*x[col[index]];
   y[_P1[warp][31]] += _P2[warp][31];}
```

Listing 4.1: setup_for_segreduce transformation sequence for shared memory reduction involving scalar expansion. Loop peeling is done subsequently to isolate updates by threads on warp boundaries

```
/*5.  Scalar Expand across indices (block, warp) for arrays _P1 and _P2
    to communicate
intermediate values for second level reduction  */
for(block=0; block < NZB; block++){
 for(warp=0; warp < NZW; warp++){
  for(by_warp=0; by_warp < DIM_X - 1; by_warp++){
   for(cidx=0; cidx < 32; cidx++){
    _P1[warp][cidx] = c_i[index];
    _P2[warp][cidx] =  A[index]*x[col[index]];
    y[_P1[warp][cidx]] += _P2[warp][cidx];
   }
  }
  for(cidx=0; cidx < 31; cidx++){
   _P1[warp][cidx] = c_i[index];
   _P2[warp][cidx] =  A[index]*x[col[index]];
   y[_P1[warp][cidx]] += _P2[warp][cidx];
  }
  _P1[warp][31] = c_i[index];
  _P2[warp][31] =  A[index]*x[col[index]];
  _P_DATA1[block][warp] = _P1[warp][31];
  _P_DATA2[block][warp] = _P2[warp][31];
  y[_P_DATA1[block][warp]] += _P_DATA2[block][warp];
 }
}
/*6.  Distribute newly introduced statements for 2 level reduction
*/
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++){
   for(by_warp=0; by_warp < DIM_X - 1; by_warp++){
    for(cidx=0; cidx < 32; cidx++){
     _P1[warp][cidx] = c_i[index];
     _P2[warp][cidx] =  A[index]*x[col[index]];
     y[_P1[warp][cidx]] += _P2[warp][cidx];
    }
   }
   for(cidx=0; cidx < 31; cidx++){
    _P1[warp][cidx] = c_i[index];
    _P2[warp][cidx] =  A[index]*x[col[index]];
     y[_P1[warp][cidx]] += _P2[warp][cidx];
   }
   _P1[warp][31] = c_i[index];
   _P2[warp][31] =  A[index]*x[col[index]];
   _P_DATA1[block][warp] = _P1[warp][31];
   _P_DATA2[block][warp] = _P2[warp][31];
  }
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++)
   y[_P_DATA1[block][warp]] += _P_DATA2[block][warp];
```

Listing 4.2: setup_for_segreduce transformation sequence for separating loop nests for first and second level reductions.

```
/*7. Scalar Expand across indices (block,warp) for shared memory
    reduction for second level gpu*/
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++){
   for(by_warp=0; by_warp < DIM_X - 1; by_warp++){
    for(cidx=0; cidx < 32; cidx++){
     _P1[warp][cidx] = c_i[index];
     _P2[warp][cidx] =  A[index]*x[col[index]];
     y[_P1[warp][cidx]] += _P2[warp][cidx];
    }
   }
   for(cidx=0; cidx < 31; cidx++){
    _P1[warp][cidx] = c_i[index];
    _P2[warp][cidx] =  A[index]*x[col[index]];
    y[_P1[warp][cidx]] += _P2[warp][cidx];
   }

   _P1[warp][31] = c_i[index];
   _P2[warp][31] =  A[index]*x[col[index]];
   _P_DATA1[block][warp] = _P1[warp][31];
   _P_DATA2[block][warp] = _P2[warp][31];
  }
 for(block=0; block < NZB; block++)
  for(warp=0; warp < NZW; warp++){
   _P3[block][warp] = P_DATA1[block][warp];//shared memory copy
   _P4[block][warp] = _P_DATA2[block][warp]; //shared memory copy
   y[_P3[block][warp]] += _P4[block][warp];
  }

```

Listing 4.3: setup_for_segreduce transformation sequence for second level reduction in shatred memory.

4.4 Results

We measure the performance of compiler-generated SpMV code on a set of matrices from the University of Florida Sparse Matrix Collection[65]. Table 4.1 lists the matrices with their number of rows, nonzeros and average nonzeros per row. The experiments use the Nvidia Tesla C2050 Fermi, which has 14 Streaming Multiprocessors with 32 cores per SM. It has 1 GB of global memory and a 64KB register file per streaming multiprocessor. We report relative performance measurements, comparing the generated code to the corresponding CUSP implementation described in [5]. The compiler generates the three versions of the code from the previous section: CSR Scalar, CSR Vector, and COO.

4.4.1 CSR Scalar

In Figure 4.4, we compare performance of the CSR Scalar code and its relative speedup over CUSP. We use three different 1D block sizes representing the number of threads

Table 4.1: A suite of unstructured test matrices.

	Matrix	N	NNZ	$\frac{NNZ}{N}$
1	cant	62,451	4,007,383	64.1
2	consph	83,334	6,010,480	72.1
3	mac_econ_fwd500	206,500	1,273,389	6.1
4	mc2depi	525,825	2,100,225	3.9
5	pdb1HYS	36,417	4,344,765	119.3
6	pwtk	217,918	11,634,424	53.3
7	rma10	46,835	2,374,001	50.6
8	scircuit	170,998	958,936	5.6
9	shipsec1	140,874	7,813,404	55.4

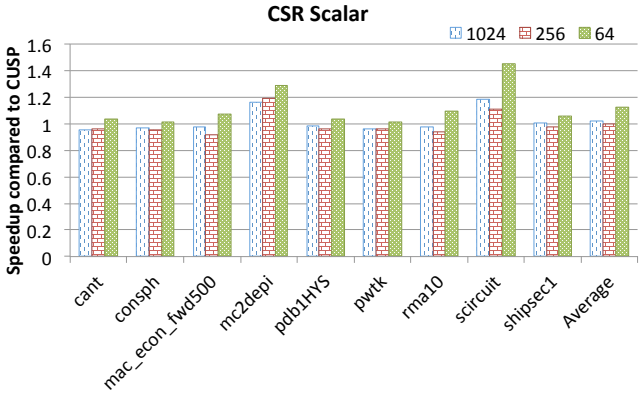


Figure 4.4: Speedup of CSR Scalar generated code with respect to its CUSP implementation.

per block: 1024, 256, and 64. In all versions, each thread computes a single row of the output vector. As shown in Figure 4.4, observe that using a block size of 64 gives the best performance for each matrix, indicating the performance of CSR Scalar code depends on the number of threads per block (block size) rather than on the matrix properties. The best performance attained by the compiler-generated code across all matrices is comparable or even exceeds the performance of the CUSP CSR Scalar implementation in some cases, with an average improvement over CUSP of 1.13X. The improvement over CUSP is presumably due to the block size; CUSP uses a fixed block size of 256 for CSR Scalar.

4.4.2 CSR Vector

Figure 4.5 compares two versions of CSR Vector and their speedup over CUSP's CSR Vector implementation. The *Base* case uses a fixed 2D block of size 32 $\times$16, with 32 threads per matrix row. Observe that performance of *Base* is close to the CUSP implementation on 6 out of 9 matrices. However it performs poorly for the matrices: *mac_econ_fwd500*, *mc2depi*, and *scircuit*. For these, the average number of nonzeros per row (see Table 4.1) is below 10, and thus there are idle resources, and unnecessary synchronization in the reduction. The CUSP library adjusts the number of threads per row based on number of nonzeros; for the compiler-generated code, we employ autotuning to identify the best 2D block size, with the additional performance gained by the best solution captured by the *Autotuned* bar. For autotuning, we vary the 2D block size (x, y) such that, $x \in \{2, 4, 8, 16, 32\}$ and $y \in \{16, 32, 64, 128, 256\}$ and $x \times y = 1024$. The pair of numbers on top of each bar represent the best 2D block size per matrix, and results improve significantly so that the average speedup over CUSP is 1.15$\times$.

4.4.3 COO

Figure 4.6 illustrates the cumulative performance improvements of applying different optimizations for COO related to improving memory bandwidth and reducing control flow, shown in comparison to CUSP. The "Base" case is a simpler compiler-generated version. The first optimization, "Indirection and padded", eliminates IF conditions that

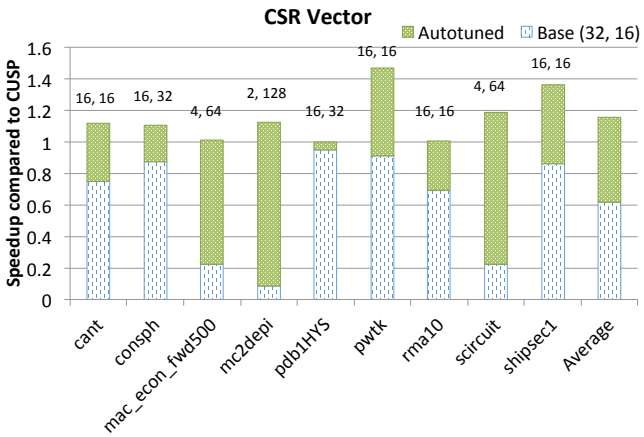


Figure 4.5: Speedup of CSR Vector generated code with respect to its CUSP implementation.

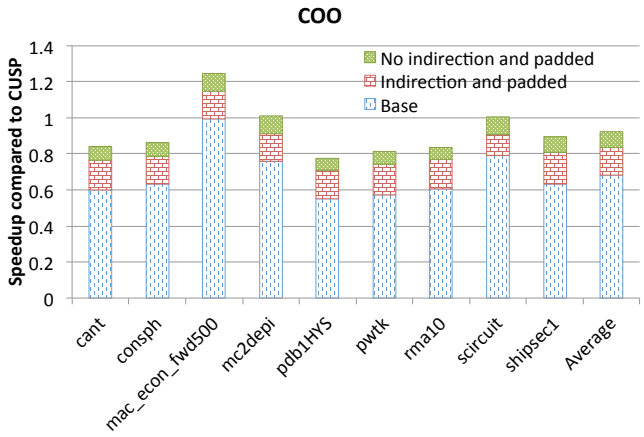


Figure 4.6: Speedup of COO generated code with respect to its CUSP implementation.

check if the array accesses are within bounds by padding the data structure with zeroed entries (_P1 in the spmv_first_level_gpu kernel, refer to Figure 4.3(f)). _P1 is padded to 64 elements even though a warp of size 32 threads access it. The reduction proceeds by checking the segment id of other threads that are *less* than it by 16,8,4,2,1 in that order. By prepadding shared memory with 32 invalid values for segment descriptors, we avoid the IF-condition check for within bounds accesses. On average, this optimization improves the performance by 15%. The removal of indirection as discussed in Chapter 3 further improves the overall performance by an additional 10% on average as indicated by the "No indirection and padded" bar. With these optimizations the generated COO code achieves 92% of the corresponding CUSP implementation. CUSP still achieves marginally better performance than the automatically-generated COO implementation. This slightly improved performance in CUSP is due to the tight coupling of reduction and the remainder of SpMV, whereas in our implementation the reduction implementation has been abstracted so as to achieve a clean separation from the core kernel; that is, a small performance loss results from the systematic compiler-based derivation.

4.5 Summary

This chapter described how the Nonaffine representations and transformations introduced in the previous chapter facilitated the derivation of high performance code variants for SpMV on the GPU. Exposing sufficient fine-grained parallelism and global memory

coalescing were introduced as high performance strategies for the GPU. Various parallelization strategies for SpMV, as in Bell and Garland [5], were reviewed and their corresponding compiler implementations were described. Representing Nonaffine loop bounds was imperative to deriving the transformation sequence for *CSR Vector*, one of the code variants. The *generalized* loop coalescing transformation's role in enabling other transformations in the derivation of the *COO* code variant for SpMV was also demonstrated. Results competitive with Nvidia's CUSP library [5] were obtained for the code variants *CSR Scalar*, *CSR Vector*, and *COO*.

CHAPTER 5

LOOP AND DATA TRANSFORMATIONS FOR SPARSE MATRIX CODE

There has been a proliferation of sparse matrix formats in recent years to improve the performance of sparse matrix applications on emergent architectures. Sparse matrix computations tend to be memory bound, where the cost of data movement dwarfs computation [3]. Some sparse matrix formats introduce a small number of zero-valued elements to the data to regularize memory accesses and simplify the generated code.

In this chapter, we introduce three new loop and data transformations for sparse matrix codes. For each of these new transformations, the compiler automatically generates inspector/executor code that reorganizes the iteration space and/or data. An advantage of this approach, is that these transformations compose with other iteration space transformations such as tiling and unroll-and-jam to further optimize the code. The transformations are sufficiently generic in that they can be utilized to systematically derive a wide range of sparse matrix representations tailored to different architectures.

5.1 Overview of approach

This section describes the new transformations *make-dense*, *compact*, and *compact-and-pad*. Figure 5.1 illustrates how the new transformations interact with each other and other transformations.

- First, *make-dense* takes as input any set of nonaffine array index expressions and introduces a guard condition and as many dense loops as necessary to replace the nonaffine index expressions with an affine access. The *make-dense* transformation enables further loop transformations on the dense loops introduced, such as skewing, shifting and tiling.

- The *compact* and *compact-and-pad* transformations are

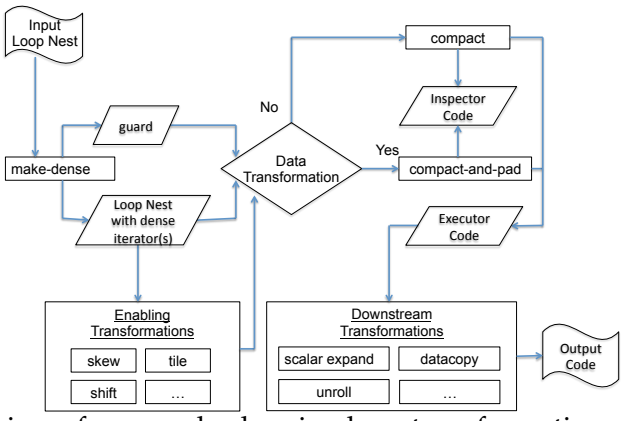


Figure 5.1: Overview of approach, showing how transformations are incorporated.

inspector-executor transformations; an automatically-generated inspector gathers the iterations of a dense loop that are actually executed, and the optimized executor only visits those iterations. The executor represents the transformed code that uses the compacted loop, which can then be further optimized by downstream loop transformations.

- In the *compact-and-pad* transformation, the inspector also performs a data transformation, inserting explicit zeros when necessary to correspond with the optimized executor.

5.1.1 Make-dense

Figure 5.2 presents before and after code templates for *make-dense*, illustrating its effects on the control flow, loop bounds, and array index expressions. The arguments of command *make-dense*(*s*0,[$idx_1, \ldots, idx_m$]) identify the statement and the index expressions to which the transformation is to be applied.

Each input index expression is replaced with a corresponding dense loop iterator, $I_{x_1}, \ldots, I_{x_m}$. These dense loops iterate over the range of their corresponding index expression and are placed immediately outside some loop I_k, where I_k is the innermost loop upon which any of the m index expressions depend. As the set of index expression values may not be a continuous sequence, and as loop iterators are continuous integer sequences, a guard surrounds the loop body to compare the new dense iterators to the associated index functions. Finally the nonaffine index expressions are replaced by a reference to the new

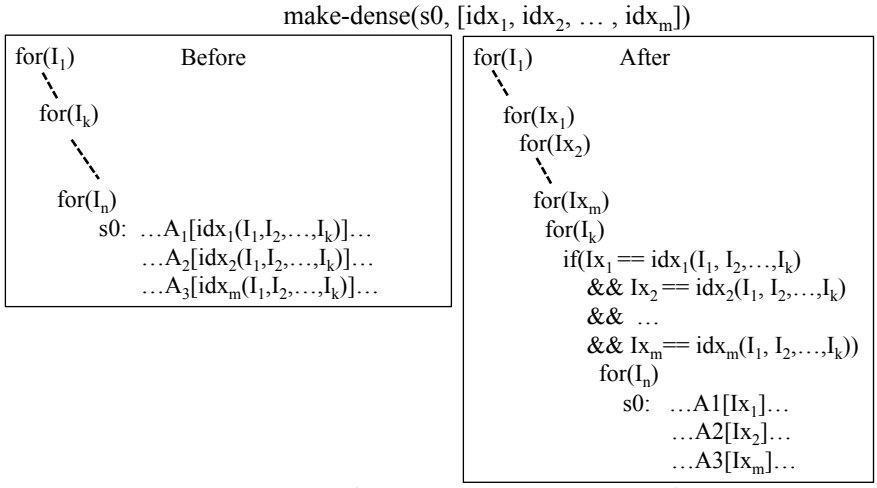


Figure 5.2: Template for the make-dense transformation.

loop iterator.

Although the transformed code after the *make-dense* transformation is executable unlike the sublimation approach in [45], typically we would not want to execute the code because it is not efficient. Specifically, the entire dense range for each nonaffine index expression passed to *make-dense* is now being visited although the guard ensures only the iterations from the original loop are executed. However, *make-dense* is still useful as an enabling transformation because it enables tiling over the range of index expressions, register tiling, and scalar replacement. In essence, *make-dense* results in a loop nest with more affine loop bounds and affine array index expressions. **Safety Test:** A conservative safety test for *make-dense* requires that the only dependences carried by loops $I_1, \ldots, I_k$ are due to reduction computations [31]. This restriction is because the new dense loops will iterate over the range of possible index expression values in order, whereas the original loop potentially employs a different order (e.g., if the nonzeros in each row `i` in Listing 5.1 are not stored in order by column). If it is possible to prove that the nonaffine array accesses are monotonically nondecreasing [41, 66], and therefore, the iterations are not reordered, then this restriction is not needed.

There are other requirements on the array index expressions and the placement of the guard introduced by *make-dense*. Any index expression where it is possible to compute its range is allowed as input to *make-dense*. Additionally, the guard depends on the loops $I_1, \ldots, I_k$ and $I_{x_1}, \ldots, I_{x_m}$ and therefore must be nested within those loops and should sur-

round the whole loop body.

5.1.2 Compact

The *compact* transformation replaces a dense loop containing a conditional guarding execution of its body with a sparse loop that only visits the iterations where the condition is true. The *compact* command takes as arguments the statement corresponding to the loop body, and I_k, the loop level whose iterations are to be evaluated against a guard. The transformed original code is called the executor, as illustrated in the before-and-after code template in Figure 5.3. The transformation also generates an inspector to pack the iteration values that satisfy the condition into array `explicit_index`, shown in Figure 5.4. The *compact* transformation is similar to guard encapsulation [67].

Each of the outer loops of I_k: $I_1, \ldots, I_{k-1}$, are represented by dimensions in `offset_index`. On iterations that satisfy the guard, `explicit_index` records the original value of I_k, and is used in place of references to I_k in the executor. Since the loop being compacted, I_k may have inner loops (e.g., loops I_{k+1} through I_{k+d} in Figure 5.3), the inspector needs to ensure that it only stores a specific value of I_k once. The *marked* variable flags the presence of a compacted loop iteration that satisfies the guard and ensures that each such iteration is counted only once. After *compact* has been applied the resulting code will have more nonaffine loop bounds and array index expressions.

An example where the *compact* transformation could be used is in the construction of

[1]The compact and compact-and-pad transformations support compaction of multiple consecutively nested loop levels. For purposes of illustration we show compact and compact-and-pad on one loop level. Multiple consecutive loop levels are treated as a single logical loop level.

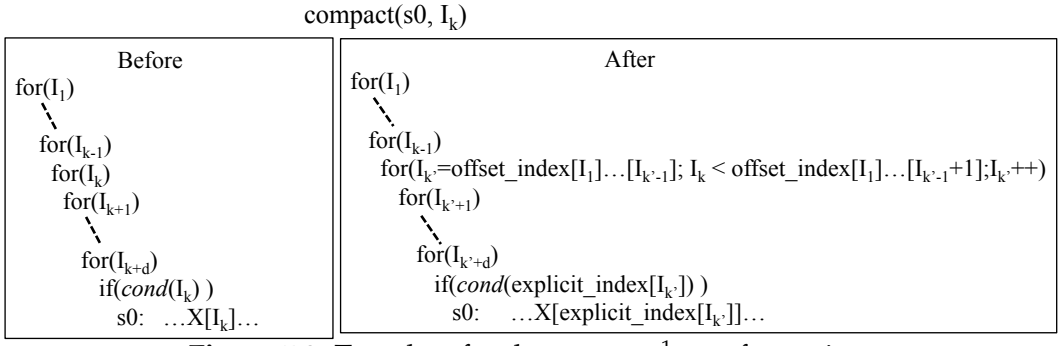


Figure 5.3: Template for the compact[1] transformation.

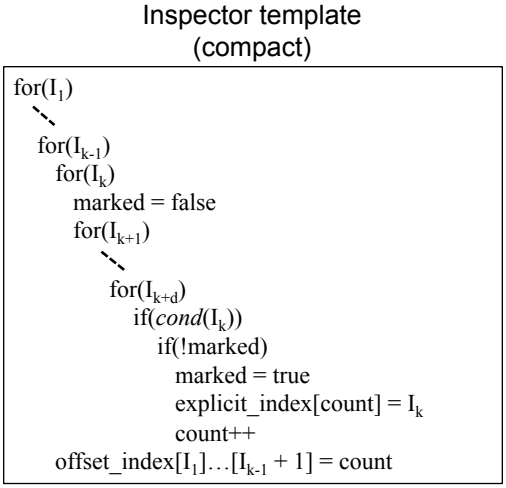


Figure 5.4: Template for the run-time inspector for compact (before optimizations in Section 5.2).

Unaligned Block Compressed Sparse Row (UBCSR) [2], Generalized Compressed Sparse Row (GCSR) [68], and Doubly Compressed Sparse Columns (DCSC) [69], where only nonempty rows or columns are stored for blocks of nonzeros in a sparse matrix. **Safety Test:** The *compact* transformation is always legal because it does not change the ordering of the iterations in the loop. It merely replaces a dense loop iterator with a sparse one that has nonaffine loop bounds. While *compact* is safe, further transformations that could have been applied to the previously affine loop bounds may no longer be applicable to that loop level.

5.1.3 Compact-and-pad

The *compact-and-pad* transformation generates a similar inspector and executor to *compact* but additionally performs a data transformation. It takes as arguments the statement corresponding to the loop body, and I_k, the loop level whose iterations are to be evaluated against a guard, and a single array `A` to which a data transformation is to be applied. The before-and-after executor code and inspector code are shown in Figure 5.5 and 5.6, respectively. Inspector generation relies on the following definition:

Definition 3. *For a given set of constraints S on a set of variables $I_1, \ldots, I_n$, Project(S, I_j) is defined as the set of reduced constraints obtained from S by eliminating every occurrence of variable I_j from all (in)equalities in S using Fourier-Motzkin elimination, i.e. every pair of inequalities of*

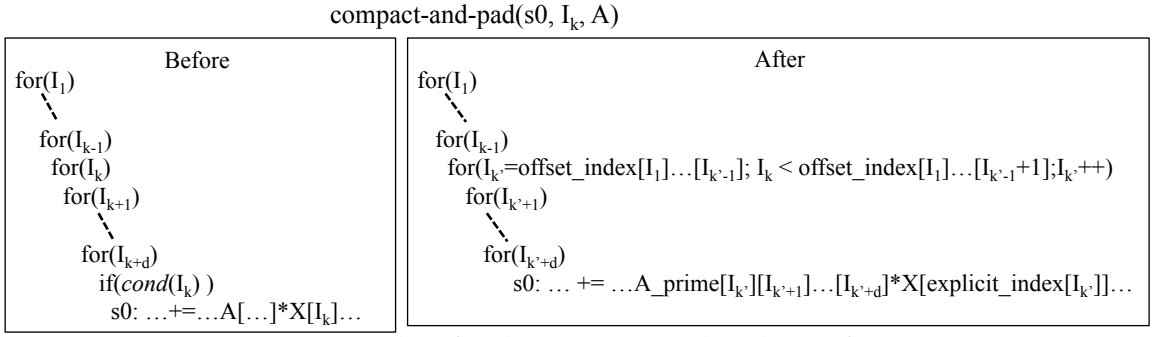


Figure 5.5: Template for the compact-and-pad transformation.

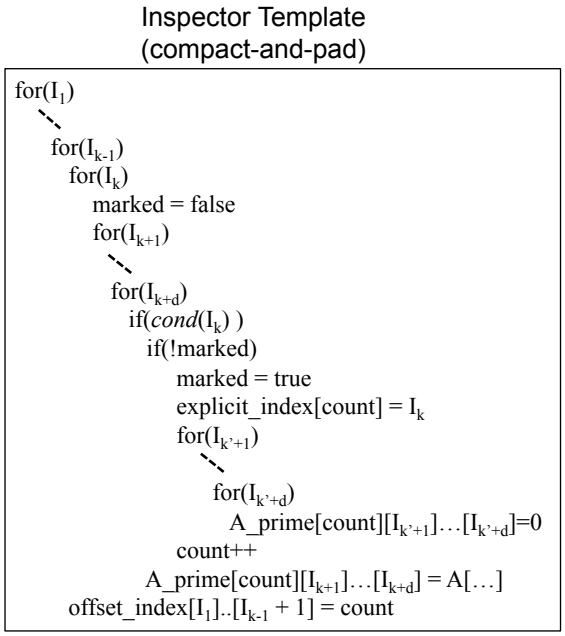


Figure 5.6: Template for the run-time inspector for compact-and-pad (before optimizations in Section 5.2).

the form $lb \leq c_1 I_j, c_2 I_j \leq ub$ *is replaced by a new inequality* $c_2 lb \leq c_1 ub$.

Conceptually, the *compact-and-pad* inspector copies the footprint of the compacted iterations associated with a specified array `A` into a transformed array `A_prime` that will be referenced in the optimized executor code. The count of the compacted loop's iterations that satisfy the guard assumes the leading dimension of the newly reorganized array, `A_prime`. When the compacted loop level is not innermost, we use the inner loops' $(I_{k+1}, \ldots, I_{k+d})$ iteration space to derive the size of `A_prime`. For each loop level j nested inside I_k, the size of the dimension corresponding to that level is computed as $ub_j - lb_j + 1$, where

$lb_j \leq I_j \leq ub_j$ and lb_j and ub_j are the respective lower and upper bounds on the loop level I_j in the set of constraints obtained from *Project*(S,I_k),...,*Project*(S,I_{j-1}). That is, the outer loop constraints are projected into each inner loop to derive the inner loops' maximum loop bounds. These bounds are then used to allocate the corresponding array dimension.

Additionally *compact-and-pad* might pad data elements with an identity value (0 in the template) in `A_prime` to effect redundant but harmless computation for associative operators. The inspector can insert an identity value, particular to the type of operation, into `A_prime` even on iterations that do not satisfy the guard. This allows *compact-and-pad* to eliminate the guard condition in the executor where *compact* would not, as illustrated in Figure 5.3 and 5.5 (c). The `marked` variable in *compact-and-pad* serves an identical function as in *compact*. **Safety Test:** Eliminating the guard condition in *compact-and-pad* is unsafe if two distinct input iteration space vectors evaluate to the same location in the array being transformed. Thus, *compact-and-pad* aborts in the inspector if two distinct data entries being inspected are mapped to the same array location. This is the injectivity criterion. Any affine transformations that manipulate the loop indices in the guard such that the injectivity criterion is violated will cause *compact-and-pad* to fail. Further, padding and eliminating the guard relies on updates involving `A_prime` to be associative and to have an identity value.

5.1.4 Example: CSR SpMV

Applying *make-dense* and a subsequent *compact-and-pad* on the SpMV code in Listing 5.1, based on the templates presented in Figure 5.2, results in the executor code shown in Listing 5.3. This code is roughly equivalent to the original CSR SpMV code.

5.1.5 Example: Traversing an unstructured mesh

The example input code in Listing 5.4 traverses over triangles in an unstructured mesh and performs various operations by accessing values associated with the three nodes of each triangular element. The code has three distinct nonaffine index expressions `n1`, `n2` and `n3`, and illustrates how *make-dense* and *compact* could be called on multiple array index expressions and loop levels simultaneously.

Three outer loops are inserted by *make-dense*, as shown in Listing 5.5, with one corresponding to each indirect reference. The execution is guarded with a logical conjunction

over the conditions corresponding to each loop level. Then, *compact* is subsequently called with all three outer loops simultaneously specified to produce the inspector in Listing 5.6. Since all loop levels for *compact* are continuous, they are treated as a single logical entity. The inspector code records the value of each iterator that satisfies the guard, one per compacted loop level. The inspector code in Listing 5.6 is further optimized as outlined in Section 5.2.2. The optimized executor code appears in Listing 5.7.

5.1.6 Example: BCSR SpMV

Now let us examine how to use these transformations to modify the matrix representation. The inspector copies from `A` to `A_prime` to introduce additional zero-valued elements. We first apply *make-dense* to the CSR input of Listing 5.1 and produce the code in Listing 5.2. To derive dense blocks of constant size $R \times C$, we can tile the output of *make-dense*, both the `i` and `k` loops, corresponding to the rows and columns of the sparse matrix.

```
1 for(i = 0; i < N; i++)
2  for(j = index[i];j < index[i+1]; j++)
3   y[i] += A[j]*x[col[j]];
```

Listing 5.1: CSR SpMV code.

```
1 for(i = 0; i < N; i++)
2  for(k = 0; k < N; k++)
3   for(j = index[i];j < index[i+1]; j++)
4    if(k == col[j])
5     y[i] += A[j]*x[k];
```

Listing 5.2: CSR SpMV after make-dense.

```
1 for(i = 0; i < N; i++)
2  for(k = offset_index[i]; k < offset_index[i+1]; k++)
3   y[i] += A_prime[k]*x[explicit_index[k]];
```

Listing 5.3: Executor for CSR SpMV.

```
1  for (e=0; e<numelem; e++){
2   ... data[ n1[e] ] ...
3   ... data[ n2[e] ] ...
4   ... data[ n3[e] ] ...
5  }
```

Listing 5.4: Triangle code.

```
for(n3i=0; n3i < N; n3i++)
 for(n2i=0; n2i < N; n2i++)
  for(n1i=0; n1i < N; n1i++)
   for (e=0; e<numelem; e++)
    if(n1i==n1[e] && n2i==n2[e] && n3i==n3[e]){
     ... data[ n1i ] ...
     ... data[ n2i ] ...
     ... data[ n3i ] ...
    }
```

Listing 5.5: Triangle after make-dense.

```
count=0
for(n3i=0; n3i < N; n3i++){
 marked_3 = false;
 for(n2i=0; n2i < N; n2i++){
  marked_2 = false;
  for(n1i=0; n1i < N; n1i++){
   marked_1 = false;
   for (e=0; e<numelem; e++)
    if(n1i==n1[e] && n2i==n2[e] && n3i==n3[e])
     if(!(marked_3 && marked_2 && marked_1)){
      marked_3  = true;
      marked_2  = true;
      marked_1  = true;
      explicit_index_1[count] = n1i;
      explicit_index_2[count] = n2i;
      explicit_index_3[count] = n3i;
      count++;
     }
  }
 }
}
```

Listing 5.6: Inspector resulting from compact for Triangle.

```
for(i=0; i < count;i++){
    ... data[ explicit_index_1[i] ] ...
    ... data[ explicit_index_2[i] ] ...
    ... data[ explicit_index_3[i] ] ...
}
```

Listing 5.7: Executor resulting from compact for Triangle.

```
for(ii = 0; ii < N/R; ii++)
 for(kk = 0; kk < N/C; kk++)
  for(i = 0; i < R; i++)
   for(k = 0; k < C; k++)
    for(j = index[ii*R + i]; j < index[ii*R+i+1]; j++)
     if(kk*C + k == col[j])
      y[ii*R + i] += A[j]*x[kk*C + k];
```

Listing 5.8: CSR SpMV after make-dense and tiling.

```
for(ii = 0; ii < N/R; ii++)
 for(kk = offset_index[ii]; kk < offset_index[ii+1]; kk++)
  for(i = 0; i < R; i++ )
   for(k = 0; k < C; k++)
    y[ii*R + i] += A_prime[kk][i][k]*x[C*explicit_index[kk] + k];
```

Listing 5.9: Executor for BCSR SpMV.

This is shown in Listing 5.8. The tiles are then the inner loops (with `j` temporarily remaining as innermost loop), and the tile controlling loops `ii` and `kk` permuted to the outermost positions. This tiling is proven safe as it does not modify the direction of the dependence on `y`. The associated transformation relation is:

$$T = \{[i,k,j] \rightarrow [ii,kk,i,k,j] | C * kk + k < N \;\&\&\; R * ii + i < N\} \quad (5.1)$$

A subsequent *compact-and-pad* at loop level `kk` produces the BCSR executor shown in Listing 5.9. The corresponding CHiLL script and inspector for BCSR are shown in Figure 5.7 and Listing 5.10, respectively.

5.2 Optimization of inspector and executor

A key consideration in automating an inspector/executor approach for sparse matrices, and a distinguishing feature of our work, is to derive high-performance inspectors that

```
make_dense(stmt,"j","k")

tile(stmt,"i", R, 1, counted)
tile(stmt,"k", C, 1, counted)

compact-and-pad(stmt,"kk","a", "a_prime")

--downstream transformations
--copy to temporary storage and
--fully unroll inner loops

datacopy(executor_stmt,"k", x)
datacopy(executor_stmt, "i", y)
unroll(executor_stmt,"k", C)
unroll(executor_stmt, "i", R)
```

Figure 5.7: CHiLL script for BCSR.

```
struct list_item{
  float data[R][C];
  int col;
  struct list_item *next;
};
struct mk{
  struct list_item *list_ptr;
};
offset_index[0] = 0;
count = 0;
struct mk marked[];
struct list_item *list=NULL;
for(ii = 0; ii < N/R; ii++){
 for(i = 0; i < R; i++)
  for(j = index[ii*R + i]; j < index[ii*R+i+1] ; j++){
   kk = col[j]/C;
   marked[kk].list_ptr = NULL;
  }
 for(i = 0; i < R; i++)
  for(j = index[ii*R + i]; j < index[ii*R+i+1] ; j++){
   kk = col[j]/C;
   k = col[j] - kk*C;
   if(marked[kk].list_ptr == NULL){
    struct list_item *new_entry =
    malloc(sizeof(struct list_item));
    for(i_ = 0; i_ < R; i_++)
     for(k_ = 0; k_ < C; k_++)
      new_entry->data[i_][k_] = 0;
    new_entry->col = kk;
    new_entry->next = list;
    list = new_entry;
    marked[kk].list_ptr = new_entry;
    count++;
   }
   marked[kk].list_ptr->data[i][k] = A[j];
  }
  offset_index[ii+1] = count;
}
```

Listing 5.10: Optimized inspector for BCSR SpMV.

perform comparably to those used in manually-tuned libraries. In particular, we want to avoid the inspector having to make several passes over the sparse matrix or introduce significant additional work not present in the original code. This section describes code generation and optimization details for both the inspector and executor.

5.2.1 Dynamic memory allocation and reduced traversals (Inspector)

The size of the new matrix representation cannot be determined statically. However, traversing the input to compute the size, as is done in OSKI, is expensive as it requires two passes over the input: one for initialization of an auxiliary data structure, and another to

traverse and count the number of nonzeros.

To minimize the number of traversals over the input, we utilize dynamic memory allocation to create a linked list of nonzero elements or blocks when they are discovered by the inspector or use static memory allocation when the size is known a priori, such as in ELL. This allows us to copy the data associated with the nonzeros while counting them in a single effective pass over the input matrix. Despite the overhead of subsequently copying the linked list into an array, this approach is faster than using two separate passes, as will be shown in Section 7.3.

5.2.2 Derivation of closed-form iterators (Inspector)

The unoptimized inspector code resulting from the *compact* and *compact-and-pad* transformations retains the loops and the guard from the input code. Since the *make-dense* command introduces additional loops to the executor and *compact* or *compact-and-pad* are typically applied after *make-dense* and other enabling transformations, the inspector's traversal over the resulting iteration space can be expensive. To reduce the overhead associated with the inspector, we replace these loop iterators with closed-form expressions wherever possible. The main idea behind this optimization is to compute the maximal set of loop iterators that can be derived from other iterators based on the guard condition for *compact*.

We utilize the rich set of polyhedral functions, such as variable projection and elimination, provided by the Omega+ library to accomplish this. The guard conditions (typically those introduced by *make-dense*) are encoded as equality constraints involving loop index variables, and all possible orderings of the loop index variables involved in the guard condition are considered to determine how to eliminate the maximum number of inner loops in the order. The guard condition is progressively updated with the eliminated variables being replaced with the corresponding expressions. A variable can be eliminated if present in the set of equalities of the current guard.

One of the patterns our algorithm detects is variables that are multiplied by some constant and then summed with a variable whose constant range is defined by the same constant (e.g., the `kk*C + k == col[j]` condition in Listing 5.8). Generally, if we have the constraints $v * c + m = e$ and $0 \leq m < c$, where c is a constant, v and m are iterators, and e is some expression, then a closed-form expression for v is $v = \lfloor e/c \rfloor$ used in conjunction

with terms being multiplied by that constant range.

For example, consider the sample guard condition `kk*C + k == col[j]`. The bounds on the `k` loop are $0 \leq k < C$ and if the `kk` loop is the candidate for elimination, the `k` loop is replaced as an existential leading to the constraints below:

$$I = \{[j, kk] \mid (\exists k : C * kk + k = col(j) \wedge 0 \leq k < C)\} \tag{5.2}$$

We observe the existence of a closed form solution for `kk` is $\lfloor col[j]/C \rfloor$ and eliminate the `kk` loop in the inspector (see lines 28 and 33 in Listing 5.10). Once `kk` is derived, its solution can be substituted into the set of constraints to yield further loop iterators such as `k`, where `k` is `col[j]-C*kk`, which would be uncovered by checking for equalities in the modified guard constraint. Hence both these loops may be eliminated from the inspector code. The optimized code with the loops and guard condition eliminated and replaced with assignments as functions of the sparse iterator is shown in Listing 5.10.

The derivation of closed form expressions used to eliminate loops also facilitates reducing the number of traversals over the input. Minimally, this optimization will eliminate the compacted loop, requiring that the scalar *marked* from Figure 5.4 be expanded to a vector, with the same size as the range of the compacted loop.

5.2.3 Elimination of loops (executor)

The additional loops introduced by the *make-dense* transformation make it imperative for the *compact* and *compact-and-pad* transformations to eliminate the additional loop(s) and/or guard introduced to minimize the run time of the executor code.

Redundant loops in the executor can be identified from considering the guard condition and the loop(s) being compacted. Let us assume that the input code for *compact* (or equivalently, *compact-and-pad*) is an `n`-deep loop nest of the form $I_1, \ldots, I_n$ and a particular loop level I_j is a candidate for compaction and elimination from the executor. The I_j loop may be eliminated if: (1) the guard is satisfied on all iterations of I_j; and, (2) an injective function F exists such that on each iteration of I_j that satisfies the guard, F maps that iteration to a unique tuple formed from the other iterators. This function is explicitly constructed in the inspector code for *compact*.

In Listing 5.11, the optimized executor code is shown with the `j` loop removed. The reference to the `j` loop in the array `A`, is now derived from `k`, using `col_inv`, which repre-

```
1 for(i = 0; i < N; i++)
2  for(k = offset_index[i]; k < offset_index[i+1]; k++)
3   y[i] += A[col_inv[k]]*x[explicit_index[k]];
```

Listing 5.11: Executor code for CSR SpMV from compact

sents the injective function F. The redundant loops being eliminated from the executor are distinct from the required loops whose closed form expressions are being derived in the inspector.

5.3 Parallel GPU implementations

We now describe two parallel implementations targeting Nvidia GPUs, which extend the work of Venkat et al. to examine implementations that require new matrix representations and will be used to compare with manually-tuned CUSP [26].

5.3.1 DIA

To uncover the diagonals that are implicit in the SpMV CSR format, the *make-dense* command is used to convert the iteration space of the sparse computation to its corresponding dense one, as we did for BCSR. After this transformation we skew the resulting dense loop `k` by `k=k-i` and permute the `i` and `k` loops to obtain the code in Listing 5.12. The transformation relations are as follows:

$$T = \{[i,k,j] \rightarrow [i,k-i,j]\} \tag{5.3}$$

$$T = \{[i,k,j] \rightarrow [k,i,j]\} \tag{5.4}$$

The outer `k` loop iterates over the diagonals that are numbered from 0 to 2*N-1, while the inner `i` loop gives the maximum possible count of the number of elements in each diagonal. However, since the matrix is sparse, the additional guard `(k+i-(N-1) == col[j])` checks for the presence of the diagonal entry. Now the *compact* command is called on the `k` loop, with the `A` matrix as argument to eliminate diagonals with nonzeros. The final executor code is shown in Listing 5.13.

The CUDA-CHiLL script for DIA is shown in Figure 5.8. Following the *make-dense* and *compact-and-pad* commands, the inner i loop is parallelized for threaded execution on a GPU. The *copy_to_shared* command copies the diagonal offset matrix or the explicit_index

```
for(k = 0; k<= 2*N-2; k++)
 for(i = max(0,N-1-k); i <= min(N-1,2*N-2-k); i++)
  for(j = index[i];j < index[i+1]; j++)
   if(k+i-(N-1) == col[j])
    y[i] += A[j]*x[k+i-(N-1)];
```

Listing 5.12: CSR SpMV after make-dense, skew and permute.

```
for(k=0; k < ub; k++)
 for(i = 0; i <= N-1; i++)
   y[i] += A_prime[k*N + i]*x[explicit_index[k]+i -(N-1)];
```

Listing 5.13: Executor for DIA SpMV.

matrix into shared memory to exploit its reuse across multiple threads, while the *copy_to_registers* command copies the output y vector to an intermediate register to accumulate the rows' dot products.

5.3.2 ELL

The ELL sparse matrix format relies on determining the maximum number of nonzero entries in a row for a sparse 2-D matrix and then extending the other rows to that length by padding with zeros. The number of nonzeros per row in the initial SpMV CSR code is implicit in the loop bounds. To make it explicit, the inner `j` loop is normalized to give an exact count of the nonzeros in each row, using a nonaffine shifting transformation as shown in Listing 5.14. The maximum row length `M` must be supplied, or can be derived by additional inspection, and is used as a tile size for the `j` loop. After *compact-and-pad*, the

```
make_dense(0,2,"k")
--enabling transformations
skew(stmt,"k",{-1,1})
permute(stmt,{"k","i","j"})

compact-and-pad(stmt,"k","a","a_prime")

--downstream transformations
permute(executor_stmt,{"i","k"})
tile_by_index(executor_stmt,{"i"},{Ti},{l1_control="ii"},
{"ii","i","k"})
tile_by_index(executor_stmt,{"k"},{Ti},{l1_control="kk"},
{"ii","i","kk","k"})

cudaize(executor_stmt,"spmv_diag_GPU",{x=N,y=N},
{block={"ii"}, thread={"i"}}, {"a_prime", "_P_DATA2"})

--shared memory and register copy --optimizations for
GPU
copy_to_shared(executor_stmt,"k","_P_DATA2",-16)
copy_to_registers(executor_stmt, "kk", "y")
```

Figure 5.8: CUDA-CHiLL script for DIA.

ELL executor is parallelized similarly to DIA, using a transposed matrix to achieve global memory coalescing and a register copy to accumulate the output results. As the nonaffine shift is somewhat unique, we include the transformation relations as follows.

$$T = \{[i,j] \rightarrow [i, j - index(i)]\} \tag{5.5}$$

$$\{[i,j] \rightarrow [i, jj, j] \mid M * jj + j < index(i+1) - index(i)\} \tag{5.6}$$

Denoting the maximum row length by a constant M, the j loop is then tiled by this amount resulting in Listing 5.15.

The `jj` loop is chosen for compaction with the `j` loop as the inner loop. We indicate that none of the row lengths exceed the constant M, by specifying it as a *known* constraint and input it to the *compact-and-pad* command. We encode it as follows:

$$\{[M][i, jj, j] \ : \ index(i+1) - index(i) < M\} \tag{5.7}$$

This means that the loop corresponding to index `jj` is a single-iteration loop and so is eliminated in the executor code shown in Listing 5.16. Further since the `jj` loop is a single-iteration loop, the arrays `A_prime` and `col_prime` are reorganized according to the outer loop index, `i` and the inner loop index `j`.

The CUDA-CHiLL script in Figure 5.9 shows the other downstream transformations applied subsequently. The loop order is permuted, and the input matrix is reordered by the CHiLL datacopy transformation based on the new loop order, so that consecutive threads access the fastest changing dimension of the input matrix. Finally the inner scalar product accumulation is done in a register by employing the *copy_to_registers* transformation.

5.4 Performance results

The transformations described in Section 5.1 are used in the transformation recipes such as in Figure 5.7 and 5.8 to derive optimized BCSR, DIA and ELL matrix represen-

```
1 for(i = 0; i < N; i++)
2  for(j = 0; j < index[i+1]-index[i]; j++)
3   y[i] += A[j+index[i]]*x[col[j+index[i]]];
```

Listing 5.14: CSR SpMV code after nonaffine shift.

```
for(i = 0; i < N; i++)
 for(jj = 0; jj <= (index[i+1]-index[i]-1)/M; jj++)
  for(j=0; j <= M - 1; j++)
   if(j <= index[i+1]-index[i]-jj*M - 1)
    y[i] += A[jj*M+j+index[i]]*x[col[j+jj*M+index[i]]];
```

Listing 5.15: CSR SpMV code after shift and tile.

```
normalize(stmt,"j")
tile_by_index(stmt,{"j"},{Tj},{l1_control="jj"},{"i","jj","j"})

shift_to(stmt,"jj",0)
shift_to(stmt,"j",0)

compact-and-pad(stmt,{"jj"},{"a_prime","col_prime"}, 0,
{"a","col"}, {1,-1},{"index_", "index__"}, M, 1)
permute(executor_stmt,{"j", "i"})

scalar_expand(executor_stmt,{"j","i"}, "a_prime")
scalar_expand(executor_stmt,{"j","i"}, "col_prime")

distribute({executor_stmt,datacopy_stmts}, "j")
fuse(datacopy_stmts,"j")

tile_by_index(executor_stmt,{"i"},{Ti},{l1_control="ii"},
{"ii","i","j"})
cudaize(executor_stmt,"spmv_ell_GPU",
{ _P_DATA4=N*M,x=N,y=N,_P_DATA3=N*M},{block={"ii"},
thread={"i"}}, {"_P_DATA4", "_P_DATA5", "n"})
copy_to_registers(executor_stmt, "j", "y")
```

Figure 5.9: CUDA-CHiLL Script for ELL Matrix Representation.

```
for(i = 0; i < N; i++)
        for(j=0; j <= M - 1; j++)
                y[i] += A_prime[i*M + j]*x[col_primel[i*M + j]];
```

Listing 5.16: Executor for ELL SpMV.

tations. These recipes include further downstream transformations as were described in Sections 5.1 and 5.3 to derive the final corresponding optimized code variants.

We use two target platforms for this experiment. The BCSR comparison focuses on optimizing for the memory hierarchy of a node in a conventional multicore architecture. For this purpose, we use an Intel i7-4770 (Haswell) CPU with 256KB L1 cache, 1 MB L2 cache, and 8MB L3 cache size. The system has 32GB of DRAM memory. The DIA and ELL comparisons look at performance on an Nvidia Tesla K20c Kepler GPU. The K20c has 13 Streaming Multiprocessors with 192 cores per SM. It has 4800 MB of global memory and a 64KB register file per streaming multiprocessor. All CPU code is compiled with the icc compiler, version 15.0.0, and all GPU code uses the nvcc compiler, version 6.5.

We compare BCSR performance to OSKI version 1.0.1h for the same set of 14 matrices used by Williams et al.[70] and obtained from the University of Florida Sparse Matrix Collection[71]. We compare DIA and ELL performance to CUSP version 0.4.0 for the same sparse matrices as in [5], structured matrices resulting from standard discretizations of the Laplacian operator. The sizes of these matrices range from 958,936 to 11.6 million nonzeros, and are therefore sufficiently large to exceed the size of the last level cache of the underlying machine.

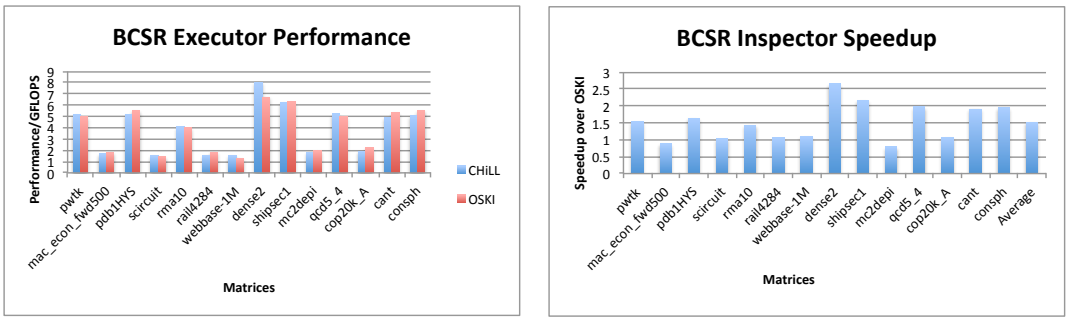
We report the absolute performance for all implementations in terms of GFlops. As sparse matrix libraries that convert to optimized matrix representations also incorporate an inspector to derive the new representation, we also compare against inspector times for OSKI and CUSP. As is done in CUSP, the compiler-generated DIA and ELL inspector code is executed sequentially on the CPU.

5.4.1 BCSR

A performance comparison of the compiler-generated BCSR code with OSKI is shown in Figure 5.10(a). We ran all configurations for block size $R \times C$ (see Listing 5.9) in the range of 1 to 8 and report the best-performing configuration for both OSKI and our compiler-generated code. The compiler-generated codes are within 1% of the performance of OSKI for the BCSR executor.

The inspector speedup, shown in Figure 5.10(b), compares the overhead of converting from CSR to the BCSR representation. On average the compiler-generated inspector is 1.5x faster than OSKI's. This is because OSKI does effectively four sweeps over the input matrix: two to compute the number of nonzero $R \times C$ blocks to allocate the memory accordingly and two more to actually fill the nonzeros in the appropriate locations of the reorganized matrix. We eliminate the first two sweeps over the input matrix by simultaneously allocating memory dynamically as a linked list of nonzero blocks and filling in the nonzeros. An additional traversal is done to reset the data structure. Finally a sweep over the nonzero blocks stored as a linked list is done to copy the data into an array layout. Hence, we accomplish the data reorganization in three sweeps as opposed to four by OSKI.

For input matrices mc2depi and mac_econ_fwd500, BCSR does not achieve any advantage over the CSR format for both OSKI and our compiler. The extra costs associated with



(a) (b)

Figure 5.10: Performance comparison of BCSR executor and inspector code with respect to OSKI.

dynamic memory allocation for every nonzero in the matrix is not amortized even by the lesser number of traversals in our inspector code compared to OSKI.

Comparing the inspector time for a particular matrix is fair only when the same $R \times C$ configuration is picked across both as larger $R \times C$ sizes lead to faster inspector times for a given matrix. In cases where the best performing configurations of the executor for both OSKI and ours were identical, such as for cant, consph, qcd5_4, and shipsec1, we observe that our inspector is uniformly faster than OSKI's, due to fewer traversals over the input matrix.

5.4.2 DIA and ELL

Figure 5.11(a) and (b) compares performance of the DIA and ELL executor code against that of the CUSP library. On average the compiler-generated code is within 5% of the performance of CUSP for both representations. The best-performing compiler-generated versions are up to 6% faster than CUSP.

We observed that as the size of the stencil increases from a 3- to 27-point stencil, the performance of the CUDA-CHiLL code variant relative to its CUSP counterpart improves. CUSP outperforms CUDA-CHiLL marginally for the 3- and 5-point stencils. This is due to a subtle difference in the two implementations. In CUSP the output vector storing the result of the matrix-vector multiply is assumed to be zero; the inner product of each vector is accumulated in a register, pre-initialized to zero, and then copied to the output vector. The code generated by CUDA-CHiLL has an additional global memory read to initialize the result vector. This read overhead is noticeable when the work per thread is relatively

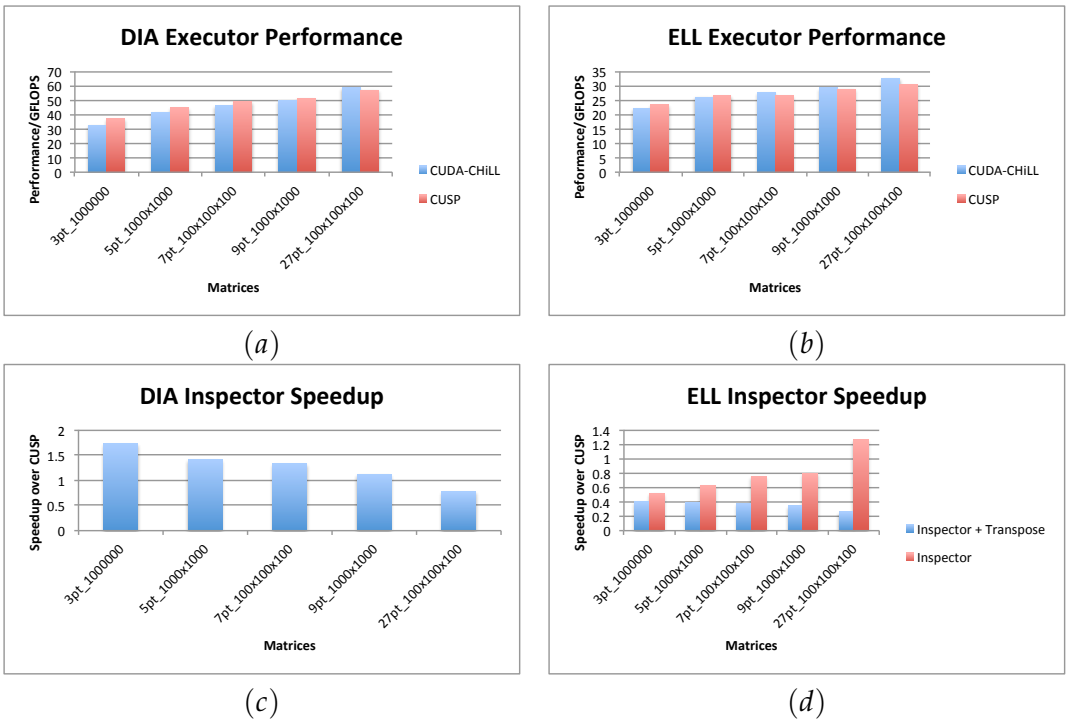


Figure 5.11: Performance comparison of DIA and ELL inspector and executor code with respect to CUSP.

small, such as for the low-order 3- and 5-point stencils. The performance advantage of the code generated by CUDA-CHiLL on the other stencils is a result of an IF-condition in the CUSP code checking if the column entries are valid in the innermost loop. CUDA-CHiLL avoids this inner conditional check by virtue of the inspector adding zeros into the data representation. DIA outperforms ELL significantly, up to a factor of 2×, because the DIA implementation reuses the elements of the `x` vector and the `offset` vector, whereas ELL cannot.

Figure 5.11(c) presents the speedup for the automatically-generated DIA inspector over CUSP, which is on average 1.27× faster. The CUSP inspector initializes the entire DIA matrix in a single pass prior to copying the nonzero entries in a separate pass, whereas the CUDA-CHiLL inspector initializes and copies the data in a single pass. Initializing and copying the data in a single pass is beneficial to exploit temporal reuse of the initialized diagonal, if it is copied into subsequently. This is the case for all matrices except the 27-point stencil where the size of the last level cache is insufficient to exploit this temporal reuse. In the case of the 27-point stencil, some of the initialized diagonals are flushed

without being reused for the data copy of the nonzeros. This result suggests that we might improve inspector time for large matrices with numerous diagonals if we could generate both versions of the inspector, and use techniques such as autotuning, learning or additional inspectors to decide which inspector is best for the input matrix.

Figure 5.11(d) examines the performance of the ELL inspector. We show two bars, labeled Inspector and Inspector+Transpose. To achieve global memory coalescing in the executor (i.e., adjacent threads accessing adjacent elements in memory), the CUSP library performs the computation on a transposed ELL matrix by declaring the ELL matrix to be in column major order. Our compiler-generated implementation must perform an additional transpose to achieve this same representation; alternatively, the generated code would be equivalent if the programmer could provide the designation of column major order to the compiler. Without the cost of the transpose, the red columns in Figure 5.11(d), the compiler-generated inspector achieves a speedup ranging from 0.52× to 1.26×. With the additional overhead of the transpose, the automatically-generated inspector code is always slower than CUSP, as the blue columns show speedups between 0.26× and 0.40×.

As the size of the matrix increases, the inspector performance relative to CUSP improves because the difference in layout is less significant for large matrices. We have identified additional optimizations to improve ELL inspector time that could be the target of future work. First, we could combine the transpose with the data layout transformation in a single sweep by permuting the loops in the CSR code prior to *compact-and-pad*, incorporating the knowledge of the fixed upper bound for the inner loop into account to make this safe. Second, we can reduce loop overhead arising from tiling by the fixed row width.

5.5 Summary

This chapter introduced three new loop and data transformations: *make-dense*, *compact* and, *compact-and-pad*. The *make-dense* transformation converts a sparse iteration space to a dense one, exposes affine loop bounds and eliminates nonaffine array access expressions. Conceptually, the dense loop iterates over the range of the nonaffine expression and a guard is introduced for correctness, as the nonaffine expression may not be continuous, while the introduced loop is a continuous integer sequence.

The dense loop can then be transformed by affine transformations. The *compact* and

compact-and-pad transformations convert a dense iteration space to a sparse one by gathering only the loop iterations that satisfy the guard introduced by *make-dense*. The *compact-and-pad* transformation effects a data transformation additionally. Both *compact* and *compact-and-pad* generate inspector/executor code to effect the code and data reorganization at run-time. Optimizations that reduce the overhead of the generated inspector and executor code were also described.

The executor code generated by *compact* and *compact-and-pad* can be further optimized by downstream transformations such as unrolling. The BCSR, ELL, and DIA sparse matrix representations were derived by composing these new transformations with other iteration space transformations. The generated BCSR code was benchmarked against OSKI on a single core CPU, while the ELL and DIA results were benchmarked against Nvidia's CUSP library on a GPU. In both cases, performance of the generated inspector and executor codes were competitive with the library implementations.

CHAPTER 6

INTEGRATION OF TRANSFORMATIONS INTO APPLICATIONS

In this chapter we demonstrate how the transformations from the previous chapter can help facilitate the derivation of highly specialized sparse matrix formats that meet the particular demands of an application. The first application is Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) which calls Sparse Matrix-Matrix (SpMM) multiply for computing the properties of light atomic nuclei [24]. In this application, the size of the matrix is so huge (approximately 430 million nonzeros) that it necessitates a sparse matrix representation with an optimized memory footprint. We describe the derivation of such a sparse matrix representation that utilizes our loop and data transformations. The second application, Stochastic Gradient Descent (SGD) [25] is a graph algorithm, where the edges of the graph are processed iteratively until some criteria of convergence is met, and on each iteration the endpoints of each edge are read and updated. SGD is challenging for parallelization because no two edges sharing an endpoint may be processed concurrently. We describe how our transformations derive efficient parallel implementations for SGD given this constraint.

6.1 LOBPCG

LOBPCG is a subspace iteration method which starts with an initial guess about the eigenvectors and refines the guess at each iteration of the solver [24]. At the heart of LOBPCG lies SpMM, which multiplies a sparse matrix with multiple dense eigenvectors. Due to the very large size of the input matrix used, the property of symmetry of the matrix is taken advantage of to store only half the matrix entries to optimize for memory footprint. Since the matrix is symmetric, SpMM using the entire matrix is accomplished by SpMM over half the symmetric matrix, followed by an additional transposed SpMM (SpMM_T) over the same half.

SpMM can be trivially parallelized using the CSR format by computing each row computation in parallel. However computing the transposed SpMM in parallel using the CSR format is difficult due to write conflicts on the output vector when the row computations are parallelized. One might observe that the Compressed Sparse Column(CSC) format might be ideal for parallelization of transposed SpMM, but then a similar problem would arise for computing SpMM using CSC.

The CSB format solves this problem by blocking the actual matrix dimensions into square blocks of size, say $\beta \times \beta$, where β is the blocking factor. It then determines the nonzeros falling in each block and stores them in the COO format in addition to storing the start and end offsets of each block. Now, SpMM can be effectively parallelized by block rows since they do not have any write conflicts and transposed SpMM can be effectively paralllelized by block column without conflicts. The CSB format is illustrated in Figure 6.1, where the actual matrix is blocked into $\beta \times \beta$ tiles. Block-column-wise parallelization for transposed SpMM is indicated by a vertical line while block-row-wise parallelization is indicated by a horizontal line. The tiles with dotted boundaries are actually not stored but

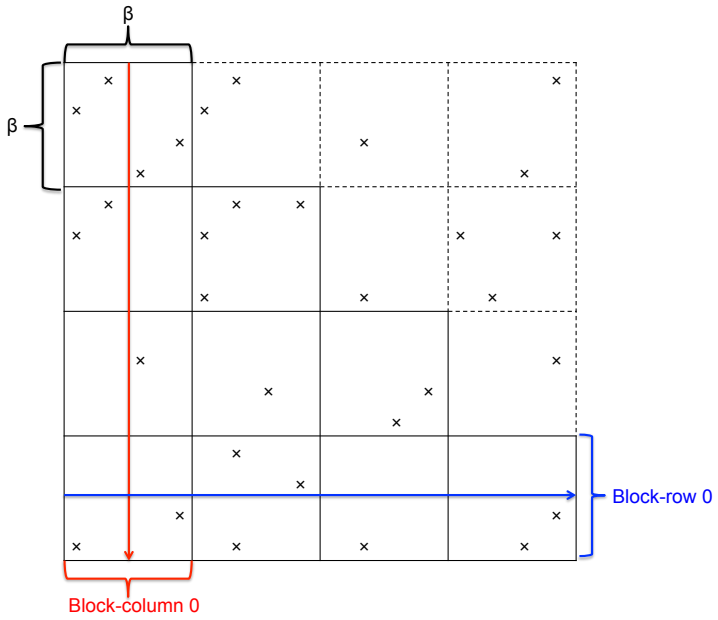


Figure 6.1: Parallelization strategy using CSB format. Nonzeros are represented by crosses. Input matrix is blocked into $\beta \times \beta$ blocks. Blocks with dotted boundaries represent symmetric portion of matrix which is not stored to reduce the memory footprint. SpMM is parallelized by block rows, while transposed SpMM is parallelized by block columns.

serve to illustrate that the actual matrix is symmetric.

6.1.1 CSB derivation

We derive the CSB format starting from the code for SpMM and transposed SpMM shown in Listings 6.1 and 6.2. The innermost loop index , `k`, iterates over the multiples right hand side vectors in SpMM. To expose the dense loops that correspond to the actual dimensions of the matrix, the *make-dense* transformation is firstly called on the SpMM code yielding the intermediate code shown in Listing 6.3. Next, tiling is applied to the two outermost loops to yield the $\beta \times \beta$ blocks in CSB. Here β is the tiling factor. The tiled code for SpMM is shown in Listing 6.4.

Finally *compact-and-pad* is applied to the consecutive third and fourth loop levels(i, l), that is the third and fourth loop levels are treated as a single logical loop level for compaction. The input sparse matrix is also reorganized by compact-and-pad into a new layout reflecting the updated traversal order of the nonzeros. Additionally the *offset_index*, *explicit_index_1* and *explicit_index_2* arrays are populated by the *compact-and-pad* inspector.The offset of each $\beta \times \beta$ block into the array of nonzeros is stored in `_P_DATA1`. Each entry of the array `_P1` corresponds to a single block, and the block's nonzeros are stored as a linked list because the size of the matrix is unknown. For each nonzero, its block is identified using the indices *ii* and *ll*. These indices specify the entry of `_P1`, whose linked list is appended with the nonzero. The row and column offsets within the block correspond to indices `i` and `l` and are stored in the linked list fields `col_[0]` and `col_[1]` respectively. The total count of nonzeros is stored in `chill_count_1` and the individual nonzero count of each block is stored in the corresponding entry in `_P1`.Once all nonzeros

```
1 for(i=0; i < n; i++)
2    for(j=index[i]; j < index[i+1]; j++)
3       for(k=0; k < m ; k++)
4          y[i][k]+= A[j]*x[col[j]][k];
```

Listing 6.1: SpMM code based on the CSR format.

```
1 for(i=0; i < n; i++)
2    for(j=index[i]; j < index[i+1]; j++)
3       for(k=0; k < m; k++)
4          y[col[j]][k]+= A[j]*x[i][k];
```

Listing 6.2: Transposed SpMM code based on the CSR format.

```
for(i=0; i < n; i++)
   for(l=0; l < n; l++)
      for(j=index[i]; j < index[i+1]; j++)
         for(k=0; k < m ; k++)
            if(l == col[j])
               y[i][k]+= A[j]*x[l][k];
```

Listing 6.3: SpMM code after make-dense.

```
for(ii=0; ii < n/beta; ii++)
   for(ll=0; ll < n/beta; ll++)
      for(i=0; i < beta; i++)
         for(l=0; l < beta; l++)
            for(j=index[ii*beta + i]; j < index[ii*beta+i+1]; j++)
               for(k=0; k < m ; k++)
                  if(ll*beta + l == col[j])
                     y[ii*beta + i][k]+= A[j]*x[ll*beta + l][k];
```

Listing 6.4: SpMM code after make-dense and tiling.

have been gathered, the offset and explicit index arrays are allocated within the memory for the right size. The data is then copied from the linked list to the arrays and, the offset of each block is updated using `_P_DATA1`. Listing 6.5 shows the inspector code for CSB.

6.1.2 Optimizations

The generated CSB code was further parallelized using OpenMP directives across block rows for SpMM and block columns for transposed SpMM. For transposed SpMM, the two outermost loops were interchanged using loop permutation so that the resulting code would be traversed by block columns.

A further optimization that reduced the memory footprint of index arrays was declaring the row and column index arrays, or *explicit_index_1* and *explicit_index_2* within a $\beta \times \beta$ block to be of 16 bit width rather than 32 bit width. To detect that the size of the index array used did not exceed the maximum allocatable size with 16 bits, the loop bounds and array access expressions were queried during *compact-and-pad* to verify the maximum possible value of the array index expression, and if they were found not to exceed the maximum allocatable size with 16 bits, were declared with a short data type.

Also, the innermost loop of SpMM does not carry a dependence, and is data parallel, and hence, is parallelized with the SIMD pragma annotation for further performance benefits. The pragma annotation is supplied via the transformation interface with the loop level for the annotation, and the code generator inserts the pragma at this loop level. The

```
for (ii = 0; ii <= 587; ii += 1)
   for (ll = 0; ll <= 589; ll += 1) {
      _P1[590 * ii + ll] = 0;
      _P_DATA1[590 * ii + ll + 1] = 0;
   }
   for (ii = 0; ii <= 587; ii += 1)
      for (i = 0; i <= 4095; i += 1)
         for (j = index[(4096 * ii + i)]; \
               j <= index[(4096 * ii + i)+1] - 1; j += 1) {
            ll = (col[j] - 0) / 4096;
            l = (col[j] - 0) % 4096;
            _P_DATA5 = ((struct a_list *)\
               (malloc(sizeof(struct a_list ) * 1)));
            _P_DATA5 -> next = _P1[590 * ii + ll];
            _P1[590 * ii + ll] = _P_DATA5;
            _P1[590 * ii + ll] -> A = 0;
            _P1[590 * ii + ll] -> col_[0] = i;
            _P1[590 * ii + ll] -> col_[1] = l;
            chill_count_1 += 1;
            _P_DATA1[590 * ii + ll + 1] += 1;
            _P1[590 * ii + ll] -> A = A[j];
         }
   for (ii = 0; ii <= 587; ii += 1) {
      if (ii <= 0) {
         _P_DATA2 = ((unsigned short *)(malloc \
                  (sizeof(unsigned short ) * chill_count_1)));
         _P_DATA3 = ((unsigned short *)(malloc \
                  (sizeof(unsigned short ) * chill_count_1)));
         A_prime = ((float *)(malloc(sizeof(float ) * chill_count_1)));
      }
      for (ll = 0; ll <= 589; ll += 1) {
         _P_DATA5 = _P1[590 * ii + ll];
         for (newVar0 = 1 - _P_DATA1[590 * ii + ll + 1]; \
                  newVar0 <= 0; newVar0 += 1) {
            _P_DATA2[_P_DATA1[590 * ii + ll] - newVar0] = \
               _P_DATA5 -> col_[0];
            _P_DATA3[_P_DATA1[590 * ii + ll] - newVar0] = \
               _P_DATA5 -> col_[1];
            A_prime[(_P_DATA1[590 * ii + ll] - newVar0) * 1] = \
               _P_DATA5 -> A;
            _P_DATA5 = _P_DATA5 -> next;
         }
         _P_DATA1[590 * ii + ll + 1] += _P_DATA1[590 * ii + ll];
      }
   }
```

Listing 6.5: SpMM inspector code.

final parallelized codes for SpMM and SpMM_T, containing SIMD and OpenMP pragmas are shown in Listings 6.6 and 6.7, respectively. OpenMP pragmas are inserted in much the same way as SIMD pragmas, specifying the statement number and loop level.

The full transformation sequence for the CSB derivation, involving a composition of *make-dense,compact-and-pad* and other loop transformations is shown in the CHiLL script in

```
#pragma omp parallel  private(ii,ll,i,k)
{
#pragma omp for  schedule(dynamic,1)
for(ii=0; ii < n/beta; ii++)
   for(ll=0; ll < n/beta; ll++)
      for(i=offset_index[ii][ll]; i < offset_index[ii][ll+1]; i++)
#pragma simd
         for(k=0; k < m ; k++)
            y[ii*beta + explicit_index_1[i]][k]+= A[i]*x[ll*beta +
    explicit_index_2[i]][k];
}
```

Listing 6.6: Parallelized and optimized SpMM executor code.

```
#pragma omp parallel  private(ii,ll,i,k)
{
#pragma omp for  schedule(dynamic,1)
for(ll=0; ll < n/beta; ll++)
   for(ii=0; ii < n/beta; ii++)
      for(i=offset_index[ii][ll]; i < offset_index[ii][ll+1]; i++)
#pragma simd
         for(k=0; k < m ; k++)
            y[ii*beta + explicit_index_2[i]][k]+= A[i]*x[ll*beta +
    explicit_index_1[i]][k];
}
```

Listing 6.7: Parallelized and optimized SpMM_T executor code.

Figure 6.2. The *omp_par_for* enables the flag for OpenMP code generation. The *split_with_alignment* transformation splits the rows to an exact multiple of the tile factor, for efficient code generation that does not include IF conditions in the innermost loop. Instead, the remaining iterations from split are executed separately in a sequential manner.

6.2 Stochastic Gradient Descent (SGD)

SGD is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions [72]. SGD is utilized for computing a low-rank matrix approximation to a graph or equivalently a sparse matrix. SGD iterates over each edge of the graph and uses the edge weight on the graph to update the entries in the two low-rank dense matrices that correspond to the endpoints of the graph until their product approximates the input sparse matrix/graph.

Recommendation systems commonly use SGD to compute a complete set of metrics given an originally incomplete one. For example, a recommendation system for movies based on user preferences would represent the incomplete set of user-movie ratings as

```
source: csb_v2.c # SpMM
procedure: csb
format : rose
loop: 0

original()
remove_dep(0,1)
fuse([0,1], 2)
split_with_alignment(0,1,4096)
split_with_alignment(1,1,4096)

make_dense(0,2,k)
known(lb == 0)
known(ub == 2412565)
known(n == 2412469)

#tile outer row and col loops by 4096
tile(0,2,4096,1,counted)
tile(0,2,4096,1,counted)

#normalize tiled loops
shift_to(0,4,0)
shift_to(0,3,0)
```

```
compact(0,[3,4],[A_prime], 0, [A])

distribute([0,1,2,3], 1)
permute(1,1,[2,1])

#OpenMP code generation
mark_omp_threads(0,[0])
mark_omp_threads(1,[0])
mark_omp_threads(2,[0])
mark_omp_threads(3,[0])

# simd code generation
mark_pragma(0,4, simd)
mark_pragma(1,4, simd)
mark_pragma(2,3, simd)
mark_pragma(3,3, simd)

#set number of OpenMP threads
omp_par_for(1,1,8)

known(index_ < index__)
known(m > 1)
```

Figure 6.2: CHiLL script for SpMM based on the CSB format.

edges in a graph, where each edge exactly connects one user and 1 movie. SGD can then be utilized to approximate the complete set of user-movie ratings. Recommender systems such as the Netflix challenge, utilize the SGD algorithm [73].

6.2.1 Parallelizing SGD using diagonal schedules

Parallelizing SGD is a challenging problem as the processing of each edge involves updating the feature vector associated with both its endpoints. This implies a required synchronization between edges sharing the same users or movies for correct updates to the shared feature vectors.

A variety of available online and offline parallelization schemes for SGD have been explored on the GPU by Kaleem et al. [25]. The online schemes explicitly synchronize at run-time using atomic instructions for locking the feature vectors associated with an edge. The offline schemes compute *matchings*, where no two edges in a matching share any endpoints. The graph is partitioned in this way into a sequence of maximal matchings, which are then scheduled with no need of online synchronization within a matching, but with synchronization required across matchings. The adjacency matrix representation of the graph depicted in Figure 6.3 is shown in Figure 6.4. To build conflict-free schedules

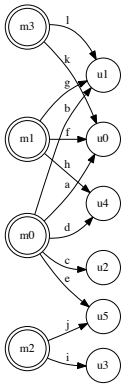


Figure 6.3: SGD graph representation

	m0	m1	m2	m3
u0	a	f		k
u1	b	g		l
u2	c			
u3			i	
u4	d	h		
u5	e		j	

Figure 6.4: Adjacency matrix representation of graph

efficiently, we can exploit the matrix representation of the graph to build matchings.

In a matrix representation, items along the diagonal have non-overlapping end points and can be processed concurrently. Different diagonals must be serialized, however. The diagonal matchings schedules rely on an *inspector* to collect edges along the diagonals of the matrix representation of a graph and an *executor* to schedule each diagonal concurrently.This parallelization is similar in principle to that used in wavefront crosssbar arbiters [74], which is an asynchronous extension to the symmetric crossbar designs of Tamir [75]. The design exploits the diagonal direction of wave propagation for parallelism.

As long as the structure of the graph does not change, this inspector need only be run once. The generation of these schedules can be performed by the compiler. We derive these diagonal schedules using a transformation sequence similar to that for the DIA representation used for SpMV.

In this section, we describe two diagonal schedules as well as their implementation via our compiler. The diagonal matchings schedules represent an instance of the compiler-

driven inspector-executor [76] implementations that can be automatically generated for sparse matrix codes [3]. We show how the serial code, shown in Listing 6.9 can be transformed by the compiler to diagonal matchings schedules that can be executed on the GPU.

Assume we are given a matrix with the number of movies, M, represented as rows, and the number of users,U, represented as columns. The maximum number of diagonals is $|M| + |U| - 1$ and the maximum width of a diagonal will be the number of columns. In our example, 4 movies (rows) and 6 users (columns) result in $4 + 6 - 1 = 9$ diagonals with the longest diagonal containing 4 entries. The complete list of diagonals, starting from the bottom is given in Figure 6.5(a). Empty entries are discarded resulting in *compressed* diagonals. Figure 6.5(b) shows the same matrix organized by block diagonals. Each block represents 2 adjacent movies and users and blocks of the same color belong to the same block diagonal.

The Diag version has the advantages of being cheap computationally to convert to by an inspector and that it is conflict-free. The block diagonal schedule, BlkDiag can be advantageous as it facilitates temporal reuse of feature vectors, but the benefits must outweigh the overhead of the barrier synchronization between diagonals. We increase the granularity of work within a diagonal, and therefore reduce the frequency of barrier synchronization in the block diagonal version. These two version are detailed in Sections 6.2.2 and 6.2.3, respectively. The recipes, which describe the corresponding series of transformations to be applied to the compiler, are described in Figure 6.6 and Figure 6.7, for the two diagonal matchings schedules.

D	Diagonal
d_0	(k)
d_1	$(-, l)$
d_2	$(f, -, -)$
d_3	$(a, g, -, -)$
d_4	$(b, -, i, -)$
d_5	$(c, -, -, -)$
d_6	$(-, h, j)$
d_7	$(d, -)$
d_8	(e)

(a) Diagonals.

	m0	m1	m2	m3
u0	a	f		k
u1	b	g		l
u2	c			
u3			i	
u4	d	h		
u5	e		j	

(b) 2×2 block diagonal schedule.

Figure 6.5: Diagonal matchings schedules for the sample input.

```
make_dense(stmt,"j")
skew(stmt,"k",{-1,1})
permute(stmt,{"k","i","j", "l"})

compact-and-pad(stmt,"i","ratings","ratings_prime")

tile_by_index(executor_stmt,{"i"},{Ti},{l1_control="ii"},{"k","ii","i","l"})

scalar_expand_by_index(executor_stmt,{"i"}, "err", SHARED_MEM,…)
scalar_expand_by_index(executor_stmt,{"i","l"}, "RHS",
SHARED_MEM,…)

cudaize(executor_stmt,"sgd_mdimDiag_GPU",…,{block={"ii"},
thread={"l", "i"}}, {"ratings_prime"})

reduce_by_index(executor_stmt, {"tx"}, "segreduce_warp", …)

datacopy_privatized(executor_stmt, "ty", "_P_DATA2", {"ty", "tx"})
```

Figure 6.6: CUDA-CHiLL script for Diag code variant.

```
make_dense(stmt,"j")

tile_by_index(stmt,{"k"},{Tk},{l1_control="kk"},{"kk", "i","k","i", "j", "l"})
tile_by_index(stmt,{"i"},{Ti},{l1_control="ii"},{"ii", "kk", "i", "k", "j", "l"})

skew(stmt,"kk",{-1,1})
permute(stmt,{"kk","ii","i", "k", j", "l"})
compact-and-pad(stmt,"ii","ratings","ratings_prime")

tile_by_index(executor_stmt,{"ii"},{Ti},{l1_control="iii"},
{"kk","iii","ii","i","k","l"})

scalar_expand_by_index(executor_stmt,{"ii"}, "err", SHARED_MEM,…)
scalar_expand_by_index(executor_stmt,{"ii","l"}, "RHS",
SHARED_MEM,…)

cudaize(executor_stmt,"sgd_blkDiag_GPU",…,{block={"iii"}, thread={"l",
"ii"}}, {"ratings_prime"})

reduce_by_index(executor_stmt, {"tx"}, "segreduce_warp", …)

datacopy_privatized(executor_stmt, "ty", "_P_DATA2", …)
datacopy_privatized_by_ref(executor_stmt, "k", user_refs, …)
datacopy_privatized_by_ref(executor_stmt, "i", movie_refs, …)
```

Figure 6.7: CUDA-CHiLL script for BlkDiag code variant.

6.2.2 Diagonal (Diag schedule)

Diag exploits the parallelism within a single edge by processing the update to the feature vector in parallel. This ordering also achieves global memory coalescing to the accesses to the feature vector across threads. We launch a 2-D grid of threads of dimension F by E, where F is the size of feature vector (e.g., 16 floats), and E is the number of edges to be processed in a kernel call. A thread (i, j) processes the i^{th} component of the feature vectors of the end points for edge j.

The full SGD code is shown in Listing 6.8 which is abstractly represented in Listing 6.9. For Diag, after the sequence of make-dense, skew, permute and shift transformations has been

```
for(i=0; i < n; i++){
 for(j=index[i]; j < index[i+1]; j++){
  //Statement s0
  err   = -ratings[j];//edge weight
  for(k=0; k < SGD_FEATURE_SIZE; k++)
   err  += fv[i][k]*fv[col[j]][k];
  for(k=0; k < SGD_FEATURE_SIZE ; k++){
   //source feature vector update
   fv[i][k]        -= step_size *
    (err * fv[col[j]][k] + SGD_LAMBDA*fv[i][k]);
   // destination feature vector update
   fv[col[j]][k] -= step_size *
    (err * fv[i][k] + SGD_LAMBDA*fv[col[j]][k]);
  }//end for-k
 }//end for-j
}//end for-i
```

Listing 6.8: SGD input code.

```
for(i=0; i < n; i++){
        for(j=index[i]; j < index[i+1]; j++){
                s0(i, col[j]);
  }//end for-j
}//end for-i
```

Listing 6.9: Abstracted SGD input code.

applied, the ratings matrix is traversed by diagonals in the outermost loop and by entries within the diagonal in the immediately enclosed loop as shown in Listing 6.10. The skew transformation is an affine transformation that is expressed as a mapping corresponding to the diagonal number. Entries along the same diagonal have the same value for the difference between column and row ids. The skew transformation maps the column id, uncovered from make-dense to this expression, so that the corresponding loop traverses diagonals. The permute transformation interchanges this loop to the outermost level so that we traverse diagonals in the outer loop, and entries within diagonals in the inner loop. The shift transformation shifts the diagonal number to a positive value, since entries below the main diagonal will have a negative diagonal value otherwise.

After this sequence of transformations, the iteration space contains redundancy and mimics that of the dense matrix, due to the make-dense transformation. The compact-and-pad transformation when applied to the i loop, inserts inspector code to gather only those iterations for which the guard is satisfied and derives the Diag executor where only those iterations are traversed, constructing the required index arrays, offset_index and explicit_index as shown in Listing 6.11. For the Diag code variant no padding is done by

```
for(k=0; k <= 2*N - 2; k++){
  for(i = max(0, N - 1 - k); i
        <= min(N-1, 2*N - 2 -k); i++){
    for(j=index[i]; j < index[i+1]; j++){
      if(k + i - (N-1) == col[j]){
         s0(i, k+i-(N-1));
      }//End if
    }//end for-j
  }//end for-i
}//end for-k
```

Listing 6.10: SGD code after make-dense,skew permute and shift.

```
for(k=0; k <= 2*N - 2; k++){
  for(i = offset_index[k];
     i <= offset_index[k+1]; i++){
     s0(explicit_index[i],
                 k+explicit_index[i]-(N-1));
   }//end for-i
}//end for-k
```

Listing 6.11: SGD executor code from compact.

compact-and-pad but the data in array ratings_prime is reorganized by diagonals rather than by the row dimension as in ratings.

The scalar_expand_by_index and reduce commands cause each thread to own a feature vector entry in shared memory and facilitate the subsequent shared memory reduction. datacopy_privatized exploits the reuse of index arrays repeatedly referenced in the executor by storing them in registers. cudaize designates certain loops as parallel CUDA dimensions, for parallel code generation.

6.2.3 Block-diagonal (BlkDiag)

The BlkDiag schedule reduces the size of the matrix by blocking along both dimensions. This reduced matrix has a reduced number of diagonals – if the movies are blocked by a factor R, and the users by a factor C, then the total number of diagonals in the BlkDiag schedule is $|M|/R + |U|/C - 1$.

A diagonal schedule obtained after 2×2 blocking is shown in Figure 6.5(b). There are now only 4 diagonals with each block consisting of 4 edges. Figure 6.8 illustrates the reorganization of the graph edges according to the block diagonal that owns each edge in the corresponding adjacency matrix representation of the graph by the inspector.

Our implementation assigns one thread to each block. Within a block, the same set of

```
void build_blk_diag_schedule(Graph g){
DiagSet d;
EdgeSet edges[];
for(Edge e: g){
 diag = d.insert(e.user/C -e.movie/R + movies/R -1);
 edges[diag][e.movie%R][e.user%C].insert(e.movie, e.rating);
 }
}
```

Figure 6.8: Build the schedule for BlkDiag.

movies and users are used repeatedly and the feature vectors corresponding to those rows and columns are cached in registers or GPU shared memory.

The transformation recipe for BlkDiag varies from Diag by requiring tiling of the two outermost loops after make dense and then applying skew,permute and shift on the tile-controlling loops *ii* and *kk*. Conceptually the tiling yields 2-D blocks which the *ii* loop iterates over and the above transformations uncover the block diagonal iteration space. In this case, compact-and-pad evaluates the guard on the entire 2-D loop nest encompassed by the compacted loop level and as long as any iteration in the enclosed 2-D iteration space satisfies the guard, it copies the entire block, padding on iterations that do not satisfy the guard.

In addition to the shared memory reduction and register optimizations for Diag the recipe for BlkDiag also copies the feature vector elements that are repeatedly reused for each movie and user within the 2-D block into registers. The datacopy_privatized_by_ref transformation is invoked once for the feature vector elements involved with the users and the second time for those involved with the movies for the specific 2-D block.

6.3 Performance results

We compare the performance of the compiler generated SpMM code using the CSB format with the manual implementation from Aktulga et al. [24]. The CSB experiments were run on an Intel i7-4770 (Haswell) CPU with 256KB L1 cache, 1 MB L2 cache, 8MB L3 cache size and 32GB of DRAM memory. We compiled our codes using the Intel C compiler version 15.0.0. The manual implementation was compiled using the Intel Fortran compiler version 15.0.0.

The SGD experiments where conducted on the Nvidia Tesla K40c and were compiled using nvcc version 6.5. In these experiments we compare the performance of both the

inspectors that construct the diagonal schedule, as well as the executors, which are the actual SGD computations based off the diagonal schedules.

6.3.1 SpMM

Figure 6.9 shows the performance of SpMM and SpMM_T using the CSB sparse matrix representation, parallelized with 8 OpenMP threads and using a blocking factor of 4096 (β =4096). As observed from Figure 6.9, the combined SpMM and SpMM_T implementation parallelized with OpenMP pragmas achieves 0.77× of the performance of the manual implementation. With the additional compiler optimization that detects that the size of the index arrays can be represented with 16-bit arrays or the *short* data type, the performance improves to approximately 0.83× of the performance of the manual implementation. Finally the SIMD pragma for vector execution of the innermost loop improves the performance to achieve a speedup of 1.06× over the manual implementation.

6.3.2 SGD

In Figure 6.10 we compare the speedup of the BlkDiag variant's inspector and executor code relative to the Diag variant. The inspector for the BlkDiag code variant is marginally slower than the Diag code variant. On average it is 0.96× as fast as the Diag inspector. This is due to the extra loop overhead in the BlkDiag inspector as it has a more nested looping structure. However the BlkDiag executor is 1.26× faster than the Diag executor on average and on all inputs. This is due to the fewer number of global synchronizations incurred

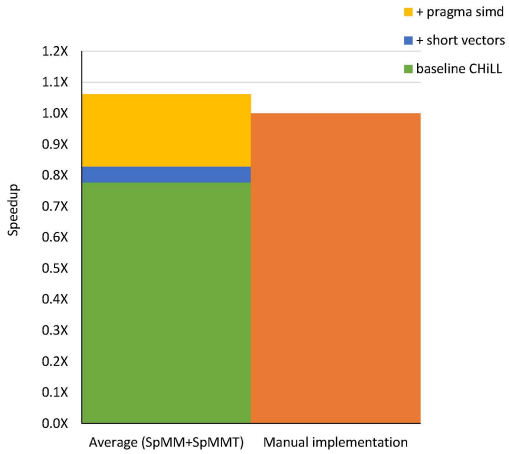


Figure 6.9: SpMM and SpMM_T results using CSB.

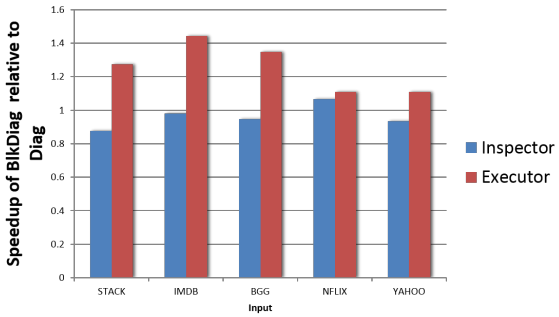


Figure 6.10: Inspector and executor performance of BlkDiag and Diag.

by BlkDiag. Further since the executor code is called multiple times in the iterative SGD algorithm and the inspector is only called once to derive the modified representation, it is more important to optimize the executor code than the inspector code in this context. In this light, the marginal slowdown of the BlkDiag inspector compared to Diag is tolerable given its significant speedup in the executor.

6.4 Summary

In this chapter, we observed how application requirements can influence the choice of the sparse matrix representation used. In LOBPCG the large size of the input matrices necessitate a sparse matrix representation that has less memory and indexing overhead, hence only half the matrix is stored by exploiting symmetry. However this necessitates an extra SpMM_T operation over the matrix, and parallelizing both SpMM and SpMM_T requires a specialized representation. The CSB sparse matrix representation is ideal for these criteria and is realized using our loop and data transformations. Both SpMM and SpMM_T are parallelized effectively using CSB by parallelizing across block rows and block columns respectively. Further indexing overhead is eliminated at compile time by detecting that indices of nonzeros within a block may be stored using 16-bit vectors. The compiler generates code parallelized with OpenMP pragmas, and vector instructions with SIMD pragmas to outperform the manual implementation.

SGD is a graph algorithm and represents the graph's corresponding adjacency matrix as a sparse matrix. SGD is an iterative algorithm that repeatedly scans the edges of the graph and updates the endpoints on the edges. No two edges that share an endpoint may be processed in parallel. This requires traversing the matrix by its diagonals, and we utilize loop and data transformations to derive diagonal representations for SGD similar to deriv-

ing the DIA sparse matrix representation for SpMV. The Diag code variant is conceptually simpler to implement than the BlkDiag variant which blocks the matrix and derives block diagonals. the BlkDiag variant is observed to have better executor performance than the Diag variant as it has less global synchronizations. The inspector for BlkDiag is marginally slower, but this can be tolerated as the inspector is only called once to derive the sparse matrix representation, while the executor is called multiple times by the iterative SGD algorithm.

CHAPTER 7

AUTOMATED WAVEFRONT PARALLELIZATION OF SPARSE CODES

Computations such as the forward Gauss Seidel relaxation shown in Listing 7.1 are challenging to parallelize due to the presence of loop-carried or cross-iteration dependences on the outermost loop. Conventional wisdom suggests that parallelization of a loop level with cross iteration dependences may have diminished profitability. While this may be the case for dense computations, where there is a strict total ordering on all iterations, there might still be significant parallelism available for sparse computations due to fewer dependences. The amount of parallelism available heavily depends on the nonzero pattern of the matrix.

In this work, we automate the derivation of wavefront-parallel code for sparse matrix computations such as symmetric Gauss Seidel relaxations. Iterations belonging to a wavefront may be scheduled in parallel but require synchronization between those of other wavefronts. We do a compile-time dependence test, and if there are dependences stemming from nonaffine array subscripts, we generate parallel inspector code to inspect these accesses and enumerate the dependences at run-time. We then derive wavefronts from the dependence graph. We demonstrate the performance of the parallel inspectors and executors in a full Preconditioned Conjugate Gradient(PCG) computation, showing

```
1 for (i=0; i < N; i++) {
2  sum = b[i];
3  for (j=rowptr[i];j<rowptr[i + 1];j++) {
4   sum -= values[j]*y[colidx[j]];
5  }
6  y[i] = sum*idiag[i];
7 }
```

Listing 7.1: Gauss Seidel Code

speedup over sequential code.

7.1 Methodology

In this section, we outline the compile-time dependence analysis involving nonaffine subscripts, the subsequent parallelized inspector that actually enumerates the dependences at run-time and constructs the dependence graph explicitly, and optimizations for the generated inspector and executor code.

7.1.1 Data dependence analysis

Compile-time data dependence analysis identifies two distinct statement instances in a loop nest that may refer to the same memory location, and at least one of the statements is a write. More concretely, a dependence exists between iteration $I = [i_1, i_2, \ldots, i_N]$ and subsequent iteration $I' = [i'_1, i'_2, \ldots, i'_N]$ (i.e., $I < I'$) in a multidimensional iteration space if both of the iterations potentially access the same memory location, at least one of those accesses is a write, and the iterations lie within the loop bounds. Standard techniques are used to derive data dependence relations between two accesses to the same array; compile-time dependence analysis then analyzes these relations to prove that the dependences cannot be satisfied, but otherwise must assume a dependence. For instance, consider the Gauss-Seidel code in Listing 7.1. Compile-time dependence analysis is imprecise in this case because the array reference involves an indirect reference through an index array whose values are not available until runtime (e.g., `y[colidx[j]]` in Listing 7.1).

In this work, we automatically generate a runtime inspector to perform dependence testing involving such indirection. During compile-time dependence testing, the compiler uses uninterpreted functions to represent information that will not be known until run-time, where an uninterpreted function $f()$ has the property that if $x = y$ then $f(x) = f(y)$. For example, the uninterpreted function $colidx()$ represents the index array `colidx[]` for the Gauss-Seidel code in Listing 7.1. We assume that the outermost loop level is chosen for wavefront parallelization. By capturing all constraints on i_1 and i'_1 and having the inner loop iterators be existentially quantified, the data dependence relation expresses all pairs of iterations of the outermost level involved in a dependence. At runtime, the inspector explicitly constructs the dependence graph to connect all such pairs of iterations where

a dependence exists. For Gauss-Seidel, the runtime inspector that would be generated directly from this dependence relation would result in two separate loop nests (one for each conjunction) and each loop nest would be four deep (i, i', j, j'), as shown in Listing 7.2.

7.1.2 Optimized inspectors

The simplified data dependence relations described in Section 7.1.1 specify the constraints that have to hold for every pair of iterations, I and I', involved in a dependence as a mixture of affine relations and nonaffine relations involving uninterpreted functions. All such pairs of iterations are connected by an edge in the dependence graph, which the inspector constructs at runtime.

To generate the inspector code, we utilize polyhedra scanning to enumerate all values of I and I' that satisfy the constraints and utilize the polyhedral statement macro interface to specialize for the operation to be done when a pair of such values are identified. Specifically, when a dependence is found between two iterations of the outer loop, those iterations are connected in the dependence graph with an edge (note the `connectFrom()` and `connectTo()` functions in Listing 7.3).

Listing 7.3 illustrates the inspector and executor code generated when the input to the compiler is the Gauss-Seidel example from Listing 7.1. In Listing 7.3, the optimized inspector uses the functions `connectFrom()` and `connectTo()` to introduce dependences for values of i and $colidx(j')$ that are equal. The `connectFrom()` and `connectTo()` functions populate the dependence graph, `A`, with edges. The dependence graph is represented as each iteration having a set of incoming and outgoing dependence edges and then another pass creates an adjacency list representation with just incoming edges.

```
1  // Naive Inspector Code
2  //   Input: Dependence relations
3  //   Output: Dependence Graph(A)
4  for(ip=0; ip<=m-1; ip++){
5   for(jp=rowptr[ip]; jp<rowptr[ip+1];jp++){
6    for(i=0; i<=m-1;i++){
7     for(j=rowptr[i]; j<rowptr[i+1];j++){
8      if(col[jp] == i){
9       if(i < ip){connectTo(A,i,ip);}
10      else if(ip < i){connectFrom(A,ip,i);}
11           }}}}}
```

Listing 7.2: Naive Inspector Code for Gauss-Seidel.

```
// Inspector Code
//   Input: Dependence relations
//   Output: Dependence Graph(A)
#pragma omp parallel for
for(ip=0; ip<=m-1; ip++){
 for(jp=rowptr[ip]; jp<rowptr[ip+1]; jp++){
   i = colidx[jp];
   if(i < ip)      { connectTo  (A,i,ip); }
   else if(ip < i) { connectFrom(A,ip,i); }
}}

// Level Set Construction (Inspector Code)
//   Input: Dependence Graph(A)
//   Output: Level Sets(levelSetBoundaries)
deriveLevelSets(A, &levelSetBoundaries,
                &num_levels);

// Executor Code After
//  Reordering(Parallelized)
for(i=0; i<num_levels; i++){
 #pragma omp parallel for
 for(j=levelSetBoundaries[i];
     j<levelSetBoundaries[i+1]; j++){
  sum = b[j];
  for (k=rowptr[j]; k<rowptr[j+1]; k++){
   sum -= values[k] * y[colidx[k]];
  }
  y[j] = (sum * idiag[j]);
}}
```

Listing 7.3: Example of compiler output.

The code generator is able to improve the generated inspector code by performing the following three optimizations.

7.1.2.1 Unnecessary loop removal optimization

Data dependence analysis creates data dependence relations where array index expressions must be equal for a dependence to exist between two iterations of the loop being wavefront parallelized (e.g., the equality constraint $i = col(j')$ for the Gauss-Seidel example). Utilizing polyhedra scanning as the underlying methodology for enumerating dependences has a major advantage that any time an iterator can be expressed as a function of other iterators, the code generator will *not* generate a loop for that iterator (e.g., the iterator i in the Gauss-Seidel example). For the Gauss-Seidel example, this reduces the original need for a four deep nested loop to a two-deep nested loop only containing the iterators i' and j', eliminating the i loop.

7.1.2.2 Loop fusion

There are two conjunctions in the data dependence relation input to the code generator. This would typically lead to two loop nests in the inspector, one for each conjunction. We modified the code generator so that it is able to determine that the data dependence relation conjuncts are the same except for the ordering constraints on the outer loop iterators i and i'. Therefore the loops to traverse the data dependences and produce the data dependence graph can be fused and those ordering constraints placed into the loop body to select which edge to insert into the graph.

7.1.2.3 Parallelization

Since it is critical that the generated inspector is efficient to reduce runtime overhead, the code generator parallelizes the inspector, ensuring that the outer loop of the inspector is fully parallel. This is possible because of what is known about the structure of the data dependence relations and the inspector code after loop fusion.

Listing 7.3 shows an example of the structure of the inspector code for Gauss-Seidel. Note that there will always be some outer loop in the inspector and that outer loop will be either i or i' from the data dependence relation (in Listing 7.3 it is i', which is represented as `ip` in code). There will possibly be some other loops internal to the outer loop. However, the parallelization is not concerned with these loops and how many of them there are. In the innermost loop body, there will be guarded calls to `connectFrom()` and `connectTo()`. The code generator will ensure that the `connectFrom()` call always has the outer inspector loop iterator passed in as the source of the data flow dependence edge and the `connectTo()` passes it as the target. This results in the outgoing and incoming edges data structure only being updated based on the outer loop iterator. Since each outer loop iteration is only updating its own data structures, the outer loop of the inspector can be parallelized.

7.1.3 Parallelized executors

At run-time, once the wavefronts are constructed, all the iterations belonging to a level set can be scheduled in parallel without any dependence violation, but synchronization across wavefronts is required to ensure correctness.

In Listing 7.3 we see the generated executor for the barrier synchronization variant. The

code is parallelized with OpenMP directives. The array `levelSetBoundaries[]` denotes the set of tasks that belong to each level set, and which may be scheduled concurrently. The `deriveLevelSets()` function reorders the sparse matrix data structure so that the order of rows in the `rowptr[]` index array match the order of iterations within each level set. Since the size of the level sets may vary, we require the extra index array to record the start and end of each level set.

7.2 Implementation

A new transformation *sparse_partition* was developed in CHiLL to partition a particular loop level, typically the outermost one, into wavefronts of irregular length. The start and end of each irregular partition is represented using index arrays, which are represented using uninterpreted functions in CHiLL. The *sparse_partition* transformation is called twice by *sparse_wavefront* to derive a 2-level hierarchical irregular tiling as shown in Figure 7.1. The *reorder_by_inspector* transformation derives a permutation over the loop level being organized by the inspector into wavefronts. The *reorder_data* transformation removes the indirection due to the permutation by reordering the data according to the iteration space permutation. The transformations prepare the iteration space of the executor code. For the barrier synchronization variant of the code, we use the OpenMP code generation capabilities of ROSE. For the point-to-point synchronization, the compiler generates SpMP's specialized post-and-wait synchronization primitives as shown in Figure 7.2. We use *omp_par_for* and *mark_omp_parallel_region* to indicate loop levels to be designated as parallel loops and regions respectively for OpenMP code generation.

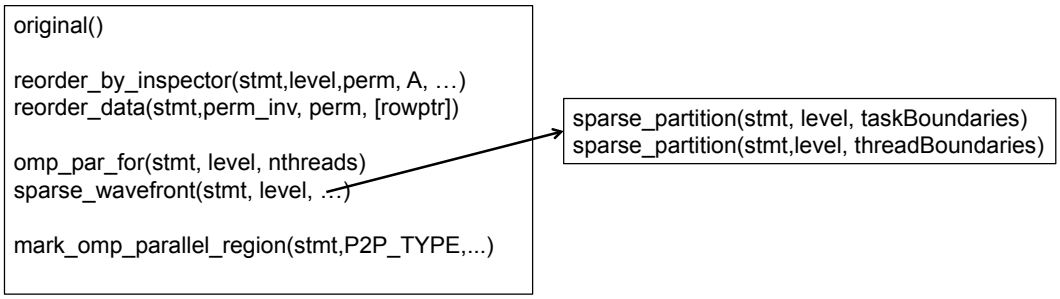


Figure 7.1: CHiLL script for point-to-point wavefront paralllelization.

```
#pragma omp parallel  private(sum,tid,t6,t8,t4) num_threads(12)
{
   int nthreads;
   tid = omp_get_thread_num();
   nthreads = omp_get_num_threads();
   //….Initiialization            (not shown)
#pragma omp barrier
{
     for (t4 = threadBoundaries[tid]; t4 <= threadBoundaries[tid+1] - 1; t4 += 1) {{
        SPMP_LEVEL_SCHEDULE_WAIT;
         for (t6 = taskBoundaries[t4]; t6 <= taskBoundaries[t4+1] - 1; t6 += 1) {
              t8 = rowptr[t6];
               sum = b[t6];
               for (t8 = rowptr[t6]; t8 <= rowptr[t6+1] - 1; t8 += 1)
                    sum -= (values[t8] * y[colidx[t8]]);
               t8 = rowptr[t6+1];
               y[t6] = (sum * idiag[t6]);
          }
      }
      SPMP_LEVEL_SCHEDULE_NOTIFY;
    }
  }
 }
```

Figure 7.2: Compiler generated point-to-point wavefront parallel code.

7.3 Experimental results

In these experiments, we evaluate the performance of the generated code for the Preconditioned Conjugate Gradient (PCG) that has been used in a prior study [55]. The PCG benchmark uses ILU0 as a preconditioner. ILU0 factorization is computed once to find L and U matrices such that L*U is close to A, subject to L and U having non-zeros at the same locations as the lower and upper triangular parts of A, respectively [77].

This accelerates the convergence of the subsequent iterative conjugate gradient calculation. We use forward and backward Gauss-Seidel relaxations for the solver, which will execute each iteration. The compiler generates two distinct implementations of the synchronization: (1) an OpenMP barrier between level sets; or, (2) a point-to-point synchronization between level sets (Post and Wait in SpMP). We use as input a representative set of matrices from the University of Florida sparse matrix collection [71], listed in Table 7.1.

We measure performance gains on a compute node of Edison at NERSC. The compute node is equipped with a dual-socket Intel Ivy Bridge generation Xeon E5-2695 v2 running at 2.4 GHz and with 64 GB of DDR3 1866 MHz main memory. Each socket has 12 cores and a shared LLC with capacity 30 MB. Each core has a private 32 KB L1 data cache and a private 256 KB L2 cache. For our experiments, we use only 12 cores in one of the two

Table 7.1: Input matrices sorted in order of increasing parallelism

Matrix	Rows	Nonzeros	Parallelism
tmt_sym	726,713	5,080,961	1.00
nd24k	72,000	28,715,634	6.28
crankseg_2	63,838	14,148,858	14.53
offshore	259,789	4,242,673	75.28
Hook_1498	1,498,023	59,374,451	95.92
af_shell3	504,855	17,562,051	135.57
Emilia_923	923,136	40,373,538	176.17
Flan_1565	1,564,794	114,165,372	200.49
bmwcra_1	148,770	10,641,602	204.35
Geo_1438	1,437,960	60,236,322	246.99
inline_1	503,712	36,816,170	287.67
StocF-1465	1,465,137	21,005,389	487.89
ecology2	999,999	4,995,991	500.50
G3_circuit	1,585,478	7,660,826	611.45
thermal2	1,228,045	8,580,313	992.96
apache2	715,176	4,817,870	1,078.70
parabolic_fem	525,825	3,674,625	87,637.50

sockets, thus avoiding NUMA effects and focusing the results on the efficiency of the generated code. We use 1 thread per core because hyperthreading does not speed up the sparse matrix operations we evaluate.

The resulting parallel code is compiled with icc version 16.0.0, and we compare our performance results with Intel Math Kernel Library(MKL) version 11.3.

Section 7.3.1 examines the performance of the generated Gauss-Seidel executor, and compares against sequential performance and Intel's MKL library. Additionally we look at the overhead of the inspection time for Gauss-Seidel. In Section 7.3.3, we compare overall parallel performance of the generated PCG to that of the Intel MKL library and

the manually-tuned code from [55].

7.3.1 Gauss-seidel inspector and executor performance

The parallel wavefront schedules computed for the Gauss-Seidel relaxations are reused multiple times. The inspector overhead takes into account the total time to construct the dependence graph as well as the modified breadth-first search to derive the wavefronts. This quantity is measured in terms of the time taken per iteration of the preconditioned conjugate gradient solver, which includes a forward and backward Gauss-Seidel relaxation. We report the performance of the parallelized executor code for the Gauss-Seidel relaxations in Figure 7.3. In Table 7.2 we observe the number of convergence iterations for each matrix. Evidently, the average inspection overhead is approximately 3.68% and the maximum, 32.33%, of the number of convergence iterations over the set of 17 matrices. The maximum inspection overhead is observed for *af_shell3* due to its relatively low number of iterations to convergence.

More than two thirds of the execution time of PCG is spent in the forward and backward Gauss-Seidel relaxations, the executors generated by our compiler. In Figure 7.3, we compare against the serial implementation, and Intel's MKL library.

We report performance in terms of effective memory bandwidth utilization. We measure the effective bandwidth in terms of the size of matrix and input/output vectors divided by the time taken for the symmetric Gauss Seidel relaxations. This is a useful metric to quantify how close the performance is to the machine peak because the sparse matrix operations we are evaluating are memory bandwidth bound due to their low arithmetic intensity [78].

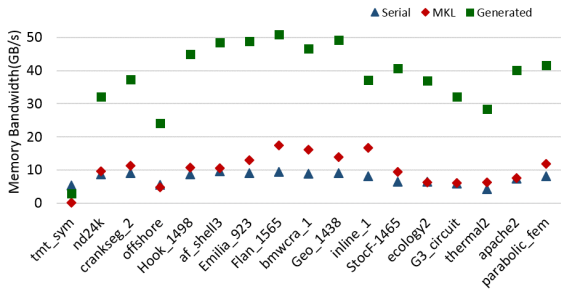


Figure 7.3: Performance of parallel Gauss-Seidel executor.

Table 7.2: Inspector overhead measured as convergence iterations.Inspector overhead# is the inspector time divided by the average time for one iteration of PCG.Inspector overhead% is this number expressed as a percentage of the total number of iterations to convergence.

Matrix	Convergence Iterations #	Inspector Overhead #	Inspector Overhead %
tmt_sym	1176	3.48	0.30
nd24k	381	22.49	5.90
crankseg_2	177	25.20	14.24
offshore	186	4.71	2.53
Hook_1498	1580	7.53	0.48
af_shell3	22	7.11	32.33
Emilia_923	411	7.75	1.89
Flan_1565	2817	8.86	0.32
bmwcra_1	1300	8.70	0.67
Geo_1438	443	7.63	1.72
inline_1	8399	15.73	0.19
StocF-1465	2493	4.56	0.18
ecology2	1791	2.92	0.16
G3_circuit	755	3.00	0.40
thermal2	1657	3.25	0.20
apache2	723	3.60	0.50
parabolic_fem	678	4.04	0.60
		Average	3.68

The median speedups achieved by the compiler generated code over the serial and Intel MKL versions are 5.26× and 3.95×, respectively.

In addition, we consider absolute performance based on the bandwidth measurements on the y-axis. As described in [55] the maximum parallel performance attainable by the Gauss-Seidel relaxation is capped by the performance of Sparse Matrix Vector Multiply (SpMV) on the corresponding matrix as they have the same number of floating point operations per nonzero and SpMV is more easily parallelized. On average the SpMV

performance measured for our configuration was approximately 50 GB/s, again matching the bandwidth on the same configuration as measured by the STREAM benchmark [79, 80]. We achieve on average 74.9% of the maximum STREAM bandwidth for the Gauss-Seidel Relaxations. Except for *tmt_sym*, which has virtually no parallelism as indicated by Table 7.1, we observe a speedup due to parallelization for the compiler generated code on all inputs.

7.3.2 Scaling

In Figure 7.4 we present the scaling of the multithreaded Gauss Seidel executor for 6 representative matrices. We measure the speedup of the multithreaded executors over the sequential implementation. Both the amount of average parallelism and the average number of nonzeros per row determine the scaling of the executors. The available work is a complex function of both the number of rows and average nonzeros per row as the wavefronts are partitioned by rows. The matrices *thermal2* and *StocF-1465* have relatively high parallelism and available work and thus achieve the highest speedups for 12 threads. The matrix *ecology2* is representative of those with relatively high parallelism but low work and they do not scale well beyond 10 threads as there is not enough work to keep all cores busy. *Flan_1565* and *Hook_1498* have sufficient work but their moderate parallelism prevents scaling beyond 10 threads. *crankseg_2* has both modest parallelism and work and so scales worse than the others.

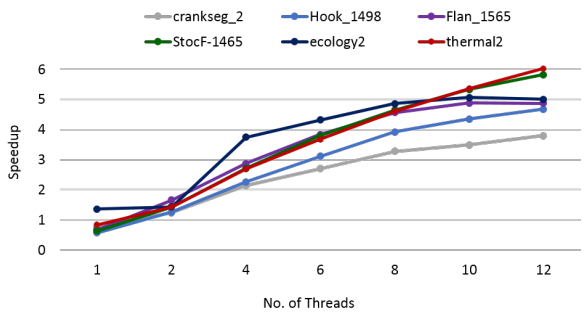


Figure 7.4: Scaling Behavior of parallel Gauss-Seidel executor.

7.3.3 Overall performance of PCG

Here we quantify the performance impact of all our optimizations for parallel PCG. For all input matrices, we demonstrate the speedup obtained over the reference sequential implementation, and compare against the Intel MKL library and manually-optimized code from [55].

We compute execution time for parallel PCG by summing the execution time of the ILU0 inspector and executor, Gauss-Seidel inspector and overall solver time. This is compared with the time for a PCG implementation utilizing sequential ILU0 factorization and Gauss Seidel relaxations. The results in Figure 7.5 show the speedup of the parallel PCG kernel over the sequential PCG implementation taking into account both the total inspector overhead and the executor times. We observe a median speedup of 2.97× over the serial implementation, for the point-to-point version. Although not shown here, the barrier version achieves a slightly more modest median speedup of 2.29×. The manually-tuned code and Intel MKL achieve a median speedup of 3.19× and 1.05× over the serial implementation. The manually tuned code has a slightly lower inspection overhead as it does not construct the dependence graph explicitly but uses the input matrix directly as the dependence graph, since the structure of the sparse matrix is equivalent to the dependence graph for the purposes of Gauss Seidel relaxations. Its inspector overhead only involves constructing the wavefronts from the input matrix.

Overall, these results demonstrate the efficacy of using an Inspector/Executor approach to parallelize PCG, and that the compiler-generated code is competitive with manually-tuned code.

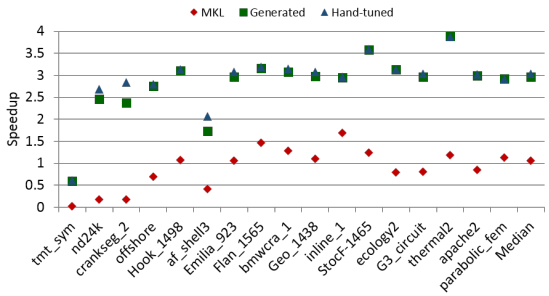


Figure 7.5: Speedup of parallel PCG over sequential PCG.

7.4 Summary

In this chapter, we demonstrated an automated wavefront parallelization methodology that utilizes both compiler and run-time optimizations for sparse matrix computations with loop-carried dependences. The constraints for loop-carried dependences are captured at compile-time and if the dependences stem from nonaffine constructs, run-time inspection code is generated to enumerate the dependences. Parallel inspectors both construct the dependence graph and derive the wavefronts. The wavefronts then execute in parallel, with synchronization across wavefronts. The point-to-point and barrier synchronization schemes were implemented for wavefront parallelization. Various optimizations for the generated inspector code, namely unnecessary loop removal using polyhedra scanning and parallelization of the dependence graph construction were discussed. The CHiLL implementation of the transformations for wavefront parallelization were detailed. The performance of the parallelized inspectors and executors was demonstrated for the PCG application.

CHAPTER 8

RELATED WORK

This section discusses prior work and focuses on the most closely-related compilers targeting sparse matrix codes.

8.1 Sparse matrix compilers

Previous work has developed compiler optimizations for sparse matrices beginning with a dense abstraction of a sparse matrix computation, as optimizations for dense matrix computations are well understood; these compilers generate sparse data representations during code generation [17–19, 81]. These compilers either incorporate a small, fixed set of matrix representations for which code generation is straightforward or rely on the user to provide implementations for accessing data in sparse formats for operations such as searching, enumeration and de-referencing. Shpeisman and Pugh [17] specify an intermediate program representation for transforming sparse matrix codes. The specification directs an underlying C++ library for efficient enumeration, permutation and scatter-gather access of nonzeros stored according to some compressed stripe storage. The Bernoulli compiler permits extension to new formats by abstracting sparse matrix computations into relational expressions that describe constraints on the iteration space and predicates to identify nonzero elements [19, 82, 83]. Using an approach similar to optimizing relational database queries, the compiler derives a plan for efficiently evaluating the relational expressions, and generates corresponding code. Gilad et al. [81] use the LL functional language for expressing and verifying sparse matrix codes with their dense analogs, under the assumption that the matrix is dense initially.

In contrast to these compilers specialized for sparse matrix computations, we have developed code and data transformations applicable to nonaffine loop bounds and subscript expressions within a polyhedral framework, in conjunction with an automatically-generated inspector. Existing nonaffine code can be optimized in this way with our com-

piler, and new matrix formats can be supported by applying the appropriate sequence of transformations. Our work is also distinguished in demonstrating that the inspection time is comparable to manually-tuned libraries.

8.2 Sublimation and guard encapsulation

The make-dense transformation is similar to sublimation presented by van der Spek et al. [45]. Sublimation requires analysis or pragmas to determine injectivity properties of the access functions so that the sublimation transformation can replace an existing loop with irregular bounds (like the inner loop in SpMV) with the dense loop. Additionally, a related idea of expanding a sparse vector into a dense vector is called *access pattern expansion* [84].

The *compact* transformation we present is related to guard encapsulation [84], which moves tests for elements into the loop bounds; in addition, *compact-and-pad* rewrites the matrix into a new representation and performs optimizations on the inspector.

Further, we have incorporated our transformations into CHiLL, which enables compositions with other polyhedral transformations and compiler-based auto-tuning within a broader context. We have designed these transformations to also generate optimized inspectors that match or beat existing hand-written inspectors in libraries.

8.3 Compiler-based approaches that support nonaffine codes

SPolly [85] extends Polly [86] with support for nonaffine constructs such as runtime specialization for program parameters with constants. SPolly achieves this by runtime profiling of parameters, identifying recurring values and replacing them with constants. One of the relevant differences between this method and our work is that, the nonaffine behavior in the program is not exposed to the underlying polyhedral framework whereas in our framework the polyhedral abstractions are extended to represent, tolerate and manipulate the nonaffine parameters, loop bounds or array access expressions.

SPolly uses runtime methods to effectively preprocess code with nonaffine behavior and convert it where possible to affine codes which can be analyzed as SCoPs(Static Control Parts), whereas we expose the runtime abstractions into the polyhedral model itself.

A related technique is employed by Sukumaran-Rajam and Clauss [87] in their Apollo compiler and runtime system. At runtime the input code is profiled and a model of the

memory accesses of the code that may involve nonaffine constructs is built. nonaffine accesses are modeled by regression equations or approximated by a regression 'tube'. A dynamic dependence polyhedron is built extending Pluto's [21] dependence testing with the linear and regression equations in a relaxed dependence polyhedron. The speculative code variant is then monitored by a decentralized verification system for any dependence violation, which is an access unpredicted by the linear and regression equations for the particular thread signaling an inter-thread interaction or dependence. A rollback is initiated upon such a violation and potentially a single threaded version of the code is executed.

Our work is distinguished from Apollo as we explicitly compute the inter-iteration dependence graph for wavefront parallelization and reorganize the code into wavefronts, whereas dependence violations would be marked by the Apollo system if such inter-iteration dependences existed and speculative execution would fail, necessitating serial execution.

8.4 Compiler-based approaches with inspector/executor extensions

Strout et al. [38] first demonstrated the feasibility of automatically generating inspectors and composing separately specified inspectors using the sparse polyhedral framework; the inspector-executor generator prototype composed inspectors by representing them using a data structure called an inspector dependence graph. We extend on this approach for automatic code generation of high performance inspector-executor code.

Rauchwerger et al. [88] use an inspector/executor methodology for wavefront parallelization of loop nests with staticallly unanalyzable access patterns. Our work uses a similar approach but with some key differences. Although they parallelize the inspector like ours, their inspector code still retains the original structure of the loop nest, which can give rise to increased time and space complexity. Moreover, during inspection they explicitly build a dependence graph for each memory location, which as they show benefits reduction and array privatization, but results in increased space consumption for inspection. In contrast, our approach simplifies the inspector via assertions [89] and results in simplified and faster inspector code.

Ravishankar et al. [49] present compiler algorithms for generating inspectors and ex-

ecutors that parallelize producer/consumer loop sequences when there are indirect memory accesses. Their parallelization is more applicable to distributed memory systems by optimizing interprocessor data communication, while ours can be applied on a single node as well. This is a different parallelization than wavefront parallelism, which is within a single loop instead of across multiple loops.

The work of McKinley [89] takes a different approach from runtime techniques: it uses user assertions about index arrays (which may otherwise be statically unanalyzable) to increase the precision of dependence testing. The assertions certify common properties of index arrays, for example, an index array can be a permutation array, monotonically increasing,monotonically decreasing strictly increasing and strictly decreasing. Our work also uses these assertions, but it uses them to improve the performance of runtime techniques, that is, it reduces the complexity of inspector code. Other data dependence analysis techniques that seek to find more parallelizable loops at compile-time include [41, 43, 90, 91].

8.5 Hand-crafted implementations for sparse matrix kernels

SpMV has been optimized for the BCSR format using OSKI [4] on CPUs. Nvidia's CUSP [5] library supports different formats such as DIA, ELL, CSR Scalar, CSR Vector ,and COO, again optimized for SpMV for GPUs.

There have been efforts on manually optimizing sparse matrix kernels with loop carried dependences, from Saltz and Rothberg's work on sparse triangular solver in 1990s [53, 92] to work on optimizing sparse triangular solve for recent NVIDIA GPUs and Intel's multicore CPUs [54, 55, 93]. Even though these manual optimizations have been successful at achieving high performance in some cases, the significant software engineering efforts involved can be affordable only for a few high profile software with manageable code complexity. This software engineering challenge is also indicated by that Intel MKL started incorporating inspector-executor optimization only very recently in their 11.3 beta version released around April 2015 [94]. One of the biggest challenges is composing the manual parallelization with other optimizations. Parallelization of kernels like sparse triangular solve needs to be composed with other optimizations such as reordering to realize its full potential as shown in [93]. Various preconditioners with different dependency pattern

can be composed with numerous choices of iterative solver algorithms [77]. Our work addresses this composability issue with the compiler.

8.6 Summary

In this chapter we discussed related work that can be categorized broadly into sparse matrix compilers, compilers that support nonaffine codes and/or inspector/executor extensions and handtuned libraries for sparse matrix computations. Early sparse matrix compilers such as MT1, SIPR and Bernoulli start off from a dense abstraction of the computation and rely on special representations to generate sparse code. We compared our *make-dense* transformation with that of sublimation [45] and the *compact* transformation with that of guard encapsulation [84]. We discussed compiler frameworks that support and/or model nonaffine code constructs. Some of these approaches abstract inspector/executor transformations. Most of these techniques are isolated inspector/executor transformations or have limited composability with polyhedral transformations. Finally we compare our techniques with manual library based approaches.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this chapter we restate the contributions of this dissertation and point to future research directions.

9.1 Nonaffine extensions

We have developed and demonstrated extensions to polyhedral code generation for supporting Nonaffine index arrays in loop bounds and subscripts. We demonstrated the robustness of this approach in applying complex sequences of new and existing transformations that integrate seamlessly with the code generation extensions.

9.2 Loop and data transformations

We have presented new transformations and an automatically-generated inspector that can be used to transform sparse matrix computations and their data representations. The compiler-generated inspector and executor code achieves performance that is comparable and sometimes exceeds the performance of popular manually-tuned sparse matrix libraries OSKI and CUSP. We see this work as an important step towards a general framework for automating transformation and data representation selection for the domain of sparse matrix computations.

9.3 Integration into applications

To clarify the scope of the effort, and the power of incorporating these transformations into an existing compiler framework, Table 9.1 shows sparse matrix formats that can be derived using our framework.

We see that a rich set of transformations are needed, both to enable the restructuring of the code and matrix representation following *make-dense*, and to generate optimized code for our two target architectures.

Table 9.1: A list of transformations performed for each variant.

	Enabling Transformations							Downstream Transformations			
	make-dense	permute	skew	shift	tile	compact	compact-and-pad	datacopy	scalar-expand	unroll	coalesce
COO				✓	✓				✓		✓
ELL				✓	✓		✓		✓		
DIA	✓	✓	✓	✓			✓				
BCSR	✓	✓			✓		✓	✓		✓	
GCSR	✓					✓					
TRI	✓					✓					
CSB	✓	✓			✓		✓				
S-DIA	✓	✓	✓		✓		✓	✓		✓	

The formats highlighted in grey, GCSR and TRI from Section 7.1, Compressed Sparse Block (CSB) [24] and S-DIA [95] can also be derived using the transformations we introduced. CSB [24] relies on blocking the dense matrix into square tiles and determining the nonzeros that fall within each tile. The nonzeros are then stored in a Coordinate (COO) format as in [26], where the row and column offsets within the tile are stored explicitly. S-DIA [95] relies on storing blocked diagonals and may be conceptualized as a hybrid between BCSR and DIA, and is derived similarly to DIA, with the addition of tiling.

9.4 Wavefront parallelization

Finally we have demonstrated how a compiler dependence analysis and parallel code generation framework can be utilized to automatically generate wavefront parallelization of sparse matrix computations. The compiler automates dependence testing for computations with loop-carried dependencs that arise in sparse computations with index array indirection and automatically generates optimized parallel inspectors and executors. Our compiler-generated inspector and executor codes outperform the corresponding computations implemented with the Inspector/Executor interface of the Intel Math Kernel Library (MKL) and nearly match that of hand-tuned code.

9.5 Contributions

To summarize our contributions are:

(1) Developing the first end-to-end polyhedral transformation and code generation system that accommodates and transforms codes with Nonaffine loop bounds and array subscripts. including inspector/executor transformations represented as Nonaffine transformations. Specifically the *generalized* loop coalescing transformation is introduced as a Nonaffine transformation that converts a loop of multiple dimensions to a

loop of a single dimension. This work was published in [26].

(2) Development of new compiler transformations, *make-dense*, *compact* and *compact-and-pad*, for codes with indirection through index arrays that facilitate code and data transformations. Code transformations include converting a loop of multiple dimensions and data transformations include padding and insertion of zero elements. The code and data transformations direct the automatic generation of high performance inspector/executor codes whose performance is competitive with manually tuned libraries. These new transformations also compose with existing polyhedral transformations. This work was published in [3].

(3) Integration of these new transformations to derive highly specialized representations for sparse matrix applications such as LOBPCG and SGD. The LOBPCG work was published in [96] and the work related to SGD was published in [25].

(4) Automated dependence testing, simplification, and inspector/executor code generation for codes with loop carried dependences resulting in parallel, high performance inspector, and wavefront-parallel executor codes. This work is to be published in [97].

9.6 Future work

We have implemented an inspector/executor methodology for both wavefront parallelization and loop and data transformations. Composing both these transformations for applying a code and data restructuring for data locality and parallelism is an interesting subject of future work. For example composing both these transformations could derive code variants such as Gauss Seidel relaxation based off the BCSR sparse matrix representation. Composed inspectors could do both the wavefront parallelization and the matrix format conversion.

Another feature that could be incorporated into our compiler framework would be supporting algorithms that do fill-in or convert a zero to a nonzero through the course of the computation. Supporting fill-in would be essential to optimizing direct methods for sparse linear systems and can be realized using the *make-dense* transformation since it exposes dense loop bounds, and potentially locations where a zero might turn into a nonzero.

The Nonaffine extensions implemented by the uninterpreted function symbol abstraction facilitate seamless polyhedral code generation for sparse matrix codes. Implementing sophisticated algorithms to minimize control flow in loop codes involving such uninterpreted function symbols is also an interesting area for future research.

Many different sparse matrix representations have been developed recently to exploit structural properties of the matrices whenever possible to improve code performance. In the future, we see the need to extend the inspector in our framework to support two new capabilities to be able to implement new sparse matrix representations. First, a number of representations require reorganizing the sparse matrix by sorting of the rows and/or columns of the input matrix, usually to expose locality [98, 99]. Other representations split the matrix into multiple parts with different characteristics, and use a different implementation for each part [100–102]. Incorporating sorting and splitting to implement hybrid schemes is the subject of future work. Further we would like to introduce abstractions for user-supplied inspectors which can then be composed within the framework.

REFERENCES

[1] W. Abu-Sufah, "A library of automatically optimized sparse matrix kernels on graphics processors." [Online]. Available: https://sites.google.com/site/sparseautotuner/matrices/hyb

[2] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, Univ. California, Berkeley, CA, USA, January 2004. [Online]. Available: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

[3] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Proc. 36th ACM SIGPLAN Conf. Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 521–532. [Online]. Available: http://doi.acm.org/10.1145/2737924.2738003

[4] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *J. Phys.*, vol. 16, no. 1, pp. 521–530, 2005.

[5] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. ACM/IEEE Int. Conf. High Performance Computing*, Nov. 2009.

[6] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Trans. on Comput.*, vol. 40, no. 5, pp. 603–612, May 1991. [Online]. Available: http://dx.doi.org/10.1109/12.88484

[7] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Run-time methods for parallelizing partially parallel loops," in *Proc. 9th Int. Conf. Supercomputing*, ser. ICS. New York, NY, USA: ACM, 1995, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/224538.224553

[8] J. Saltz, C. Chang, G. Edjlali, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. Sharma, A. Sussman, M. Uysal, G. Agrawal, R. Das, and P. Havlak, "Programming irregular applications: Runtime support, compilation and tools," *Advanced Computing*, vol. 45, pp. 105–153, 1997.

[9] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*. New York, NY, USA: ACM, May 1999, pp. 229–241.

[10] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, October 1999, pp. 192–202.

[11] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *Int. J. Parallel Prog.*, vol. 29, no. 3, pp. 217–247, 2001.

[12] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 7, pp. 606–618, 2006.

[13] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *Proc. 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP, 2013.

[14] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proc. Sym. on Principles and Practice of Parallel Programming*, 2006.

[15] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2003.

[16] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *Int. J. High Performance Computing Applications*, vol. 18, no. 1, pp. 95–114, February 2004.

[17] W. Pugh and T. Shpeisman, "Sipr: A new framework for generating efficient code for sparse matrix computations," in *Proc. 11th Int. Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, North Carolina, August 1998.

[18] A. Bik and H. A. Wijshoff, "Advanced compiler optimizations for sparse computations," in *Proc. Supercomputing*, Nov 1993, pp. 430–439.

[19] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, "Next-generation generic programming and its application to sparse matrix computations," in *Proc. 14th Int. Conf. Supercomputing*, Santa Fe, New Mexico, USA, May 2000, pp. 88–99.

[20] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," University of Southern California, Tech. Rep. 08-897, Jun. 2008.

[21] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun. 2008.

[22] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[23] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," Department of Computer Science, University of Maryland, Tech. Rep. CS-TR-3193, 1993.

[24] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Proc. 2014 IEEE 28th Int. Parallel and Distributed Processing Sym.*, ser. IPDPS, 2014.

[25] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali, "Synchronization trade-offs in gpu implementations of graph algorithms," in *Proc. IEEE 30th Int. Parallel and Distributed Processing Sym.*, 2016.

[26] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proc. Annual IEEE/ACM Int. Sym. Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 185:185–185:194. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544141

[27] C. Chen, M. Hall, and A. Venkat, "Omega+," http://ctop.cs.utah.edu/ctop/?pageid=21, 2012, [Online; accessed 15-August-2012].

[28] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Lect. Notes Comput. Sc.*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds. Springer, Sep. 2010, pp. 299–302. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/270231

[29] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 2004.

[30] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, 1996, pp. 79–103.

[31] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.

[32] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proc. Supercomputing*, Nov. 1991.

[33] H. Zhang, A. Venkat, P. Basu, and M. Hall, "Combining Polyhedral and AST Transformations in CHiLL," in *6th Int. Workshop Polyhedral Compilation Techniques (IMPACT'16)*, A. Jimborean and A. Darte, Eds., Prague, Czech Republic, Jan. 2016.

[34] C. Chen, "Polyhedra scanning revisited," in *Proc. 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation*, ser. PLDI 2012. New York, NY, USA: ACM, 2012, pp. 499–508. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254123

[35] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *ACM SIGPLAN Sym. on Principles and Practice of Parallel Programming*, Apr. 1991.

[36] F. Quilleré and S. Rajopadhye, "Generation of efficient nested loops from polyhedra," *Int. J. Parallel Prog*, vol. 28, no. 5, pp. 469–498, Oct. 2000.

[37] W. Kelly, W. Pugh, and E. Rosser, "Code generation for multiple mappings," in *Proc. Fifth Sym. on the Frontiers of Massively Parallel Computation*, Feb. 1995.

[38] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky, "An approach for code generation in the sparse polyhedral framework," *Parallel Computing*, vol. 53, no. C, pp. 32–57, April 2016.

[39] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400690

[40] W. Pugh and D. Wonnacott, "Nonlinear array dependence analysis," in *3rd Workshop Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1995.

[41] Y. Lin and D. Padua, "Compiler analysis of irregular memory accesses," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, May 2000.

[42] W. Blume and R. Eigenmann, "The range test: a dependence test for symbolic, non-linear expressions," in *Proc. Supercomputing*, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=602770.602858

[43] D. Barthou, J.-F. Collard, and P. Feautrier, "Fuzzy array dataflow analysis," *J. Parallel Distr. Comput.*, vol. 40, no. 2, pp. 210–226, 1997.

[44] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proc. Int. Conf. Compiler Construction (ETAPS CC)*, ser. LNCS. Paphos, Cyprus: Springer-Verlag, Mar. 2010.

[45] H. van der Spek and H. Wijshoff, "Sublimation: Expanding data structures to enable data instance specific optimizations," in *Proc. Int. Workshop Languages and Compilers for Parallel Computing (LCPC)*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 106–120.

[46] M. M. Strout, G. George, and C. Olschanowsky, "Set and relation manipulation for the sparse polyhedral framework," in *Proc. 25th Int. Workshop Languages and Compilers for Parallel Computing (LCPC)*, September 2012.

[47] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley, "Principles of runtime support for parallel processors," in *Proc. 2nd Int. Conf. Supercomputing*, 1988, pp. 140–152.

[48] L. Rauchwerger and D. Padua, "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proc. ACM SIGPLAN 1995 Conf. Programming language design and implementation*, ser. PLDI, 1995.

[49] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proc. Supercomputing*, November 2012.

[50] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 01, no. 01, pp. 73–95, 1989. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129053389000056

[51] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 1, pp. 123–144, Jan. 1990. [Online]. Available: http://dx.doi.org/10.1137/0911008

[52] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors impacting performance of multithreaded sparse triangular solve," in *Proc. 9th Int. Conf. High Performance Computing for Computational Science*, ser. VECPAR. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 32–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964238.1964246

[53] E. Rothberg and A. Gupta, "Parallel ICCG on a hierarchical memory multiprocessor - addressing the triangular solve bottleneck," *Parallel Computing*, vol. 18, no. 7, pp. 719–741, 1992. [Online]. Available: http://dx.doi.org/10.1016/0167-8191(92)90041-5

[54] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU," NVIDIA Corporation, Tech. Rep. 001, 2011.

[55] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver," 2014.

[56] J. Park, https://github.com/jspark1105/SpMP.

[57] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. 21st Annual Sym. Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 233–244. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584053

[58] M. Norrish and M. M. Strout, "An approach for proving the correctness of inspector/executor transformations," in *Languages and Compilers for Parallel Computing*, James Brodman and Peng Tu, Ed. Hillsboro, Oregon, USA: Springer, May 2015, pp. 131–145.

[59] N. Revol and P. Théveny, "Numerical reproducibility and parallel computations: Issues for interval algorithms," *CoRR*, vol. abs/1312.3300, 2013. [Online]. Available: http://arxiv.org/abs/1312.3300

[60] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral code generation in the real world," in *Proc. 15th Int. Conf. Compiler Construction*, Mar. 2006.

[61] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The Omega Library interface guide," University of Maryland at College Park, Tech. Rep. CS-TR-3445, Mar. 1995.

[62] M. Wolfe, *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.

[63] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural parallelization analysis in suif," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 662–731, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1075382.1075385

[64] B. Pottenger and R. Eigenmann, "Idiom recognition in the polaris parallelizing compiler," in *Proc. Supercomputing*, Nov. 1995.

[65] T. Davis, "The University of Florida Sparse Matrix Collection," *NA Digest*, vol. 97, 1997.

[66] C. Oancea and L. Rauchwerger, "A hybrid approach to proving memory reference monotonicity," in *Languages and Compilers for Parallel Computing*, ser. Lect. Notes Comput. Sc., S. Rajopadhye and M. Mills Strout, Eds., vol. 7146. Springer Berlin Heidelberg, 2013, pp. 61–75.

[67] A. Bik and H. Wijshoff, "On automatic data structure selection and code generation for sparse computations," in *Languages and Compilers for Parallel Computing*, ser. Lect. Notes Comput. Sc., U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer Berlin Heidelberg, 1994, vol. 768, pp. 57–75.

[68] S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc, "Sparse matrix vector multiplication on multicore and accelerator systems," in *Scientific Computing with Multicore Processors and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, 2010.

[69] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," *CoRR*, vol. abs/1006.2183, 2010.

[70] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009.

[71] T. Davis, "The University of Florida Sparse Matrix Collection," *NA Digest*, vol. 97, 1997.

[72] "Stochastic gradient descent," https://en.wikipedia.org/wiki/Stochastic_gradient_descent, accessed: 2016-06-14.

[73] "Netflix prize," http://www.netflixprize.com/index, http://www.netflixprize.com/index. [Online]. Available: http://www.netflixprize.com/index

[74] G. Gopalakrishnan, "Developing micropipeline wavefront arbiters," *IEEE Des. Test*, vol. 11, no. 4, pp. 55–64, Oct. 1994. [Online]. Available: http://dx.doi.org/10.1109/54.329456

[75] Y. Tamir and H.-C. Chi, "Symmetric crossbar arbiters for vlsi communication switches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 13–27, 1993, http://www.odysci.com/article/1010112992339583. [Online]. Available: http://csdl.computer.org/comp/trans/td/1993/01/l0013abs.htm

[76] R. Das, J. Wu, J. Saltz, H. Berryman, and S. Hiranandani, "Distributed memory compiler design for sparse problems," *IEEE Trans. Comput.*, vol. 44, no. 6, pp. 737–753, Jun. 1995.

[77] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[78] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[79] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[80] ——, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[81] G. Arnold, J. Hölzl, A. S. Köksal, R. Bodík, and M. Sagiv, "Specifying and verifying sparse matrix codes," in *Proc. 15th ACM SIGPLAN Int. Conf. Functional Programming (ICFP)*, ser. ICFP. New York, NY, USA: ACM, 2010, pp. 249–260.

[82] V. Kotlyar, K. Pingali, and P. Stodghill, "A relational approach to the compilation of sparse matrix programs," in *Euro-Par'97 Parallel Processing*, ser. Lect. Notes Comput. Sc., C. Lengauer, M. Griebl, and S. Gorlatch, Eds. Springer Berlin Heidelberg, 1997, vol. 1300.

[83] V. Kotlyar and K. Pingali, "Sparse code generation for imperfectly nested loops with dependences," in *Proc. 11th Int. Conf. Supercomputing*, ser. ICS, 1997.

[84] A. J. C. Bik and H. A. G. Wijshoff, "Automatic data structure selection and transformation for sparse matrix computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 2, pp. 109–126, Feb. 1996. [Online]. Available: http://dx.doi.org/10.1109/71.485501

[85] J. Doerfert, C. Hammacher, K. Streit, and S. Hack, "SPolly: Speculative Optimizations in the Polyhedral Model," in *Proc. 3rd Int. Workshop Polyhedral Compilation Techniques (IMPACT)*, Berlin, Germany, Jan. 2013, pp. 55–61.

[86] T. Grosser, A. Groesslinger, C. Lengauer, and C. S. G. Akl, "Pollyperforming polyhedral optimizations on a low-level intermediate representation," 2012.

[87] A. Sukumaran-Rajam and P. Clauss, "The polyhedral model of nonlinear loops," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 48:1–48:27, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2838734

[88] L. Rauchwerger, N. Amato, and D. Padua, "A scalable method for run-time loop parallelization," *Int. J. Parallel Programming*, vol. 23, no. 6, pp. 537–576, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF02577866

[89] K. McKinley, "Dependence analysis of arrays subscripted by index arrays," Rice University, Tech. Rep. TR91187, 1991.

[90] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proc. ACM SIGPLAN 1991 Conf. Programming Language Design and Implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/113445.113447

[91] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based array data dependences," ser. Lect. Notes Comput. Sc., K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds. Springer Berlin Heidelberg, May 1994, pp. 546–566.

[92] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 1, pp. 123–144, Jan. 1990. [Online]. Available: http://dx.doi.org/10.1137/0911008

[93] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, ser. SC. Piscataway, NJ, USA: IEEE Press, 2014, pp. 945–955. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.82

[94] "Intel Math Kernel Library Inspector-executor Sparse BLAS Routines," https://software.intel.com/en-us/articles/intel-math-kernel-library-inspector-executor-sparse-blas-routines.

[95] D. Lowell, J. Godwin, J. Holewinski, D. Karthik, C. Choudary, A. Mametjanov, B. Norris, G. Sabin, P. Sadayappan, and J. Sarich, "Stencil-aware gpu optimization of iterative solvers," *SIAM J. Scientific Computing*, pp. –1–1, 2013.

[96] K. Ahmad, A. Venkat, and M. Hall, "Optimizing lobpcg: Sparse matrix loop and data transformations in action," in *LCPC*, September 2016.

[97] A. Venkat, M. S. Mohammadi, H. Rong, R. Barik, J. Park, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *To appear in Supercomputing (SC)*, November 2016.

[98] M. Shantharam, A. Chatterjee, and P. Raghavan, "Exploiting dense substructures for fast sparse matrix vector multiplication," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 3, pp. 328–341, Aug. 2011.

[99] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. 27th Int. ACM Conf. Int. Conf. on Supercomputing*, ser. ICS. New York, NY, USA: ACM, 2013, pp. 273–282.

[100] R. Vuduc and H. Moon, "Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure," in *Proc. High Performance Computing and Communications, volume 3726 of LNCS*. Springer, 2005, pp. 807–816.

[101] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based Sparse Matrix Representation for Memory-Efficient SMVM Kernels," in *ICS*, 2009, pp. 100–109.

[102] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix-vector product computations using unroll and jam," *Int. J. High Performance Computing Applications*, vol. 18, no. 2, pp. 225–236, 2004.